PROCEDURAL REDUCTION MAPS

A Thesis
Presented to
The Academic Faculty

by

R. Brooks Van Horn III

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 3, 2007

PROCEDURAL REDUCTION MAPS

Approved by:

Dr. Gregory Turk, Adviser
College of Computing
Georgia Institute of Technology

Dr. Irfan Essa
College of Computing
Georgia Institute of Technology

Dr. Jarek Rossignac
College of Computing
Georgia Institute of Technology

Dr. Blair MacIntyre
College of Computing
Georgia Institute of Technology

Date Approved: January 5, 2007

to my wonderful, loving, and beautiful Fianceè who made this possible in all ways - I look forward to seeing You soon

PREFACE

ACM SIGGRAPH is an interesting experience. Everyone has a different opinion on it, but what I like is the ideas. Each person presenting has their own take on what is a good idea for solving a problem. They often present the benefits of their method and the difficulties with other methods. And people with conflicting ideas are the best talks because you get a much better understanding of those types of solutions.

But the truly wonderful experience is always hearing ideas and then transforming someone else's solution to their problem to an answer to your own problem.

During the summers of 2002 and 2003, I worked as a rendering intern at Pixar Animation Studios. Between the work there, and the experience at SIGGRAPH, I realized that anti-aliasing of procedural textures was a wide open field. Starting in the fall of 2003, I turned to photo-realistic rendering and never looked back.

Amazingly, even at the end of this process, I still enjoy this topic tremendously. Today there is more out there on anti-aliasing than three years ago, but it still remains open. I look forward to seeing what this thesis will bring forth in other people's minds.

ACKNOWLEDGEMENTS

I want to first off thank Eugene Zhang for providing the proper UV mapping for the objects in this work. Furthermore, Greg Turk cut UV pieces that wrapped on the object for better run-time estimation of the texture footprint.

Thank you to my lab mates: Mark Carlson (especially), Eugene Zhang, Chris Wojtan, and Huamin Wang. Their encouragement was always there (that is, if they weren't working at home that day.)

I would like my family for their encouragement and support. I wouldn't have made it through the first year without their help.

A great big thanks to Justin Jang, who had the unique place of being both a dear brother and a lab mate to me. Thanks for the prayers.

Much thanks goes to Greg Turk, for sticking with me through thick and thin, as well as being an amazing sounding board for some crazy ideas. Thanks also for keeping me grounded.

Finally, I also thank my home gathering for their support through all of this, especially at the end. Their prayers made it possible to not only endure the end, but to even experience enjoyment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Procedural textures and image textures are commonplace in graphics today, finding uses in such places as animated movies and video games. Unlike image texture maps, procedural textures typically suffer from minification aliasing. Given a procedural texture on a surface, I present a method that automatically creates an anti-aliased version of the procedural texture. The new procedural texture maintains the original texture's details, but reduces minification aliasing artifacts. This new algorithm creates an image pyramid similar to MIP-Maps to represent the texture. Whereas a MIP-Map stores per-texel color, however, my texture hierarchy stores weighted sums of reflectance functions, allowing a wider-range of effects to be anti-aliased. The stored reflectance functions are automatically selected based on an analysis of the different functions found over the surface. When the texture is viewed at close range, the original texture is used, but as the texture footprint grows, the algorithm gradually replaces the texture's result with an anti-aliased one. This results in faster development time for writing procedural textures as well as higher visual fidelity and faster rendering.

CHAPTER I

INTRODUCTION

A major goal in photo-realistic rendering is the anti-aliasing of textures. Textures are applied to most objects in a scene to enhance the object's realism. However, while rendering a texture, inaccuracies in the sampling of the texture can result in aliasing, which severely degrades realism. There are two types of textures, both of which suffer from aliasing: *image textures* and *procedural textures*. Image textures apply images to objects to enhance their realism, while procedural textures apply a programmable function to an object. Examples of procedural textures are shown in figure 1.1. This thesis has two main contributions, both centered on the anti-aliasing of procedural textures.

Chapter 3 describes in detail the first contribution of this thesis, the procedural reduction map. Chapter 4 will describe in detail the assumptions and theory behind procedural reduction maps. Chapter 5 details the second contribution, the bidirectional reflectance and normal distributions function. Both contributions will be introduced here, and chapter 2 will cover related work to these topics. Chapter 6 will cover implementation issues moving the work onto graphics hardware, leaving chapter 7 to conclude the thesis.

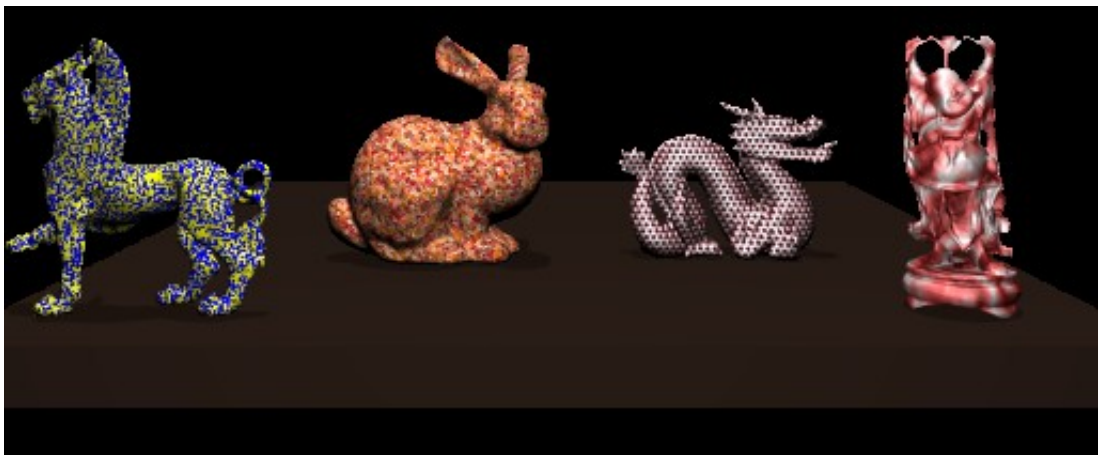


Figure 1.1: Procedural Texture Anti-Aliasing Results.

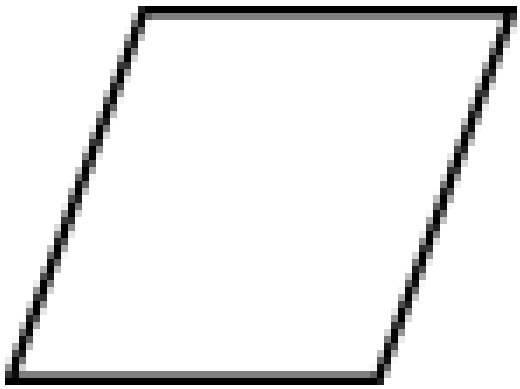


Figure 1.2: Edge Aliasing Example. Notice that the edges are not smooth, but jaggy.

1.1 Background Concepts

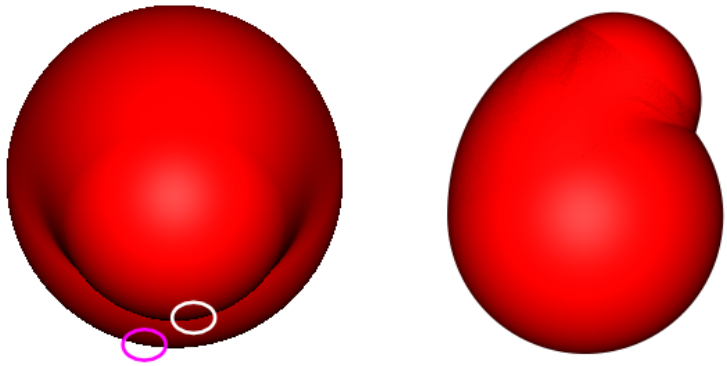
There are several concepts that need to be understood for reading this thesis. These include aliasing, reflectance functions, and textures. A brief overview of each of them follows.

1.1.1 Aliasing

Photo-realistic rendering must deal with the problem of aliasing. To fully understand aliasing, one must look at the domain of signal processing. After a signal is sampled, it needs to be reconstructed for later use. The *Nyquist frequency* is defined as the twice the frequency of the signal's maximum frequency. Sampling below this rate results in the production of a different signal. These two different signals (the original and the faulty reproduction) are indistinguishable from each other (i.e., one is an alias of the other) according to the sampling. In short, *aliasing* is deficient reproduction of a signal. In photo-realistic rendering, there are several situations that can cause aliasing to occur and each causes serious degradation of photo-realism. The rest of this section describes such cases.

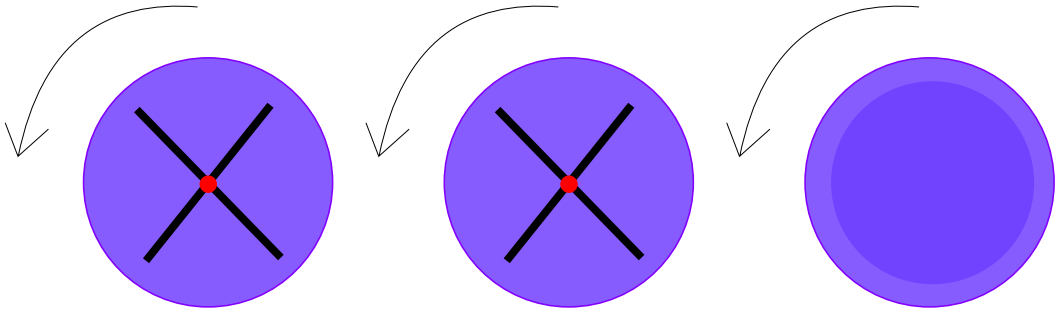
Probably the most commonly cited example of aliasing, *edge aliasing*, occurs when two different colored regions meet along an edge. Edge aliasing can be seen in photo-realistic renderings displaying a line (which, by definition, is two edges, a color discontinuity from one side of the line and another color discontinuity from the other side.) An example of edge aliasing is shown in Figure 1.2. Another case of edge aliasing occurs in textures where a portion of the texture has an edge in its pattern.

Silhouette aliasing occurs when the geometry of the object is misrepresented in the



(a) (b)

Figure 1.3: Silhouette Aliasing Example. (a) Silhouette aliasing can occur along the edge of the object (red oval) as well when the object curves back on itself (white oval). (b) Side view of object.

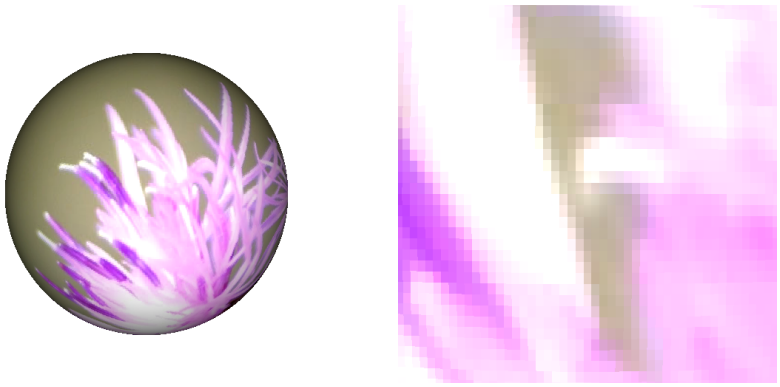


(a) (b) (c)

Figure 1.4: Temporal Aliasing Example. (a) A disk with a black X at rest. (b) A frame of an animation exhibiting aliasing. The disk is rotating at one revolution per frame and thus no change is seen. (c) A proper anti-aliased version. The X is blurred with the rest of the disk.

pixel's approximation. From the viewpoint of geometry alone, silhouette aliasing occurs when the percentage of an object's contribution to a pixel is miscalculated. Silhouette aliasing can also occur in the use of textures. If a portion of the object occludes itself, the approximation of the *texture footprint* (i.e., texture coverage as seen through the pixel) is incorrect thus leading to aliasing. Figure 1.3 demonstrates an example of silhouette aliasing.

Another situation that causes aliasing is known as *temporal aliasing*. This type of aliasing occurs across frames of an animation and can be seen from using textures as well as geometry. There are interesting real life examples of temporal aliasing that are purely geometric in nature (i.e., texture is not the source of the aliasing.) Consider a common museum exhibit: using a strobe light and a continuously dripping faucet, one can set the



(a) (b)

Figure 1.5: Magnification Aliasing Example. (a) Original sphere. (b) Magnified sphere. Notice the pixelation.

strobe light to make the drops appear to stop in mid-air or even "drop" upwards. Temporal aliasing can occur from texture use as well. Consider a texture on a sphere with a vertical line going through the poles. The sphere can be rotated around the axis fast enough (compared to the frames per second of the animation) for the line to appear stationary. Figure 1.4 exhibits temporal aliasing.

There are two forms of aliasing that are most commonly associated with textures. The first form is technically not aliasing, rather it is a problem with *signal reconstruction*, where the details of the original signal are lost after sampling. Signal reconstruction problems are sometimes referred to (and will be here) as *magnification aliasing*, and arise from the use of image textures. When the image texture is viewed too closely, the sampled function (the image) does not have enough resolution to accurately represent the original signal (the actual scene from which the image was taken), thus pixelation occurs (Figure 1.5).

The final form of aliasing commonly associated with textures is minification aliasing. *Minification aliasing* occurs when a texture is viewed from afar. Aliasing occurs when texture frequencies in screen space are higher than the Nyquist frequency given by the rendering sampling rate. This form of aliasing severely degrades photo-realism (Figure 1.6) during motion sequences and must be anti-aliased for high visual quality renderings. This is the main type of aliasing that this thesis corrects.

(a) (b) (c)

Figure 1.6: Minification Aliasing Example. The sphere is the same as the one in figure 1.5. (a) Distant sphere at 100 radii distance. (b) Distant sphere at 101 radii distance. (c) Pixels that are more than 30% different between (a) and (b).

1.1.2 Reflectance Functions

Light has several properties that are relevant for photo-realistic rendering, and, in particular, textures. Although other properties exist (such as traveling as waves), they are not as relevant to rendering, and as such will not be discussed here.

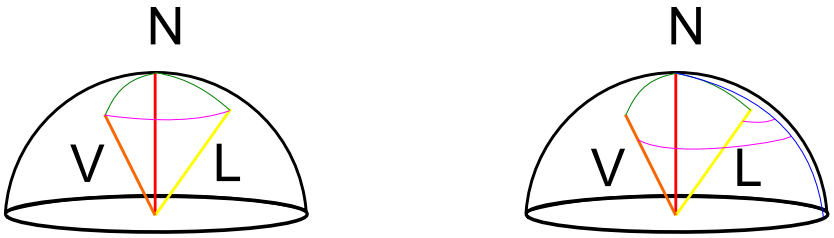
Some properties of light are that it travels along a ray, is additive (the strength of two equal rays that are coincident is equal to twice the strength of one of the rays), and the light energy entering a volume is equal to the energy absorbed inside the volume plus the energy leaving the volume.

James Kajiya introduced the rendering equation in [53]. This equation describes the relation between elements within a scene, both lights and objects, and the color perceived from a specific point in the scene. The rendering equation is

$$L(x, x') = g(x, x')[\epsilon(x, x') + \int_S \rho(x, x', x'')L(x', x'')dx''] \tag{1.1}$$

where $L(x, x')$ is the intensity of light passing from point x' to x and $g(x, x')$ is a geometric term based on the visibility between points x and x' (which is zero if x and x' are not directly visible to each other and related to the inverse distance between the patches otherwise.) $\epsilon(x, x')$ is the intensity of emitted light from x' to x, and $\rho(x, x', x'')$ is the intensity of light scattered from x'' by a patch of surface at x' to x.

No assumptions are made about the reflectance properties of objects in the scene by



(a) (b)

Figure 1.7: Reflectance Function Dimensions. The incoming light is yellow, the outgoing light is orange and the normal is the red line. Latitude arcs (pink) denote the angle between two longitude arcs. Longitude arcs (green) denote the angle between the top of the hemisphere and another point on the hemisphere.
(a) Isotropic BRDF. The intensity depends on the three angles: θ_i (the angle between the L and N), θ_o (the angle between V and N), and ϕ (the angle between L and V).
(b) Anisotropic BRDF. The blue arc represents the frame of anisotropic reference axis. The intensity depends on four angles: the previous θ_i and θ_o, as well as ϕ_i, the angle between L and the blue arc, and ϕ_o, the angle between V and the blue arc.

this equation, but $\rho(x, x', x'')$ is related to the reflectance function of a surface patch.

Although this fully describes the light interaction within a scene, it is sometimes helpful to just examine the local interaction of light with a surface. The reflectance equation (equation 1.2) describes this local interaction. Given that a direction is described by $\omega = (\theta, \phi)$, the general reflectance equation is:

$$L(x, \omega_o) = \int_{\omega_i \epsilon \Omega} L(x, \omega_i) f_r(\omega_i, \omega_o) \cos\theta_i d\omega_i \tag{1.2}$$

Reflectance functions are functions that relate the incoming light (ω_i) onto an object (x) to the outgoing light (ω_o). In photo-realistic rendering, reflectance functions are used to determine the color seen from a specific point based on the light direction (incoming light) and the viewing direction (outgoing light). Reflectance functions, at minimum, are one-dimensional in nature. However, they can be much higher in dimensionality to cover a broader range of lighting effects. Figure 1.7 relates the four most common dimensions used in reflectance functions: (θ_i, ϕ_i, θ_o, ϕ_o). Example BRDFs can be seen in figure 1.8.

A reflectance function of four dimensions or less that use only the directions of the incoming and outgoing light above the surface is often referred to as a *Bidirectional Reflectance Distribution Function*, or *BRDF*. Technically, a BRDF is not a function, but rather

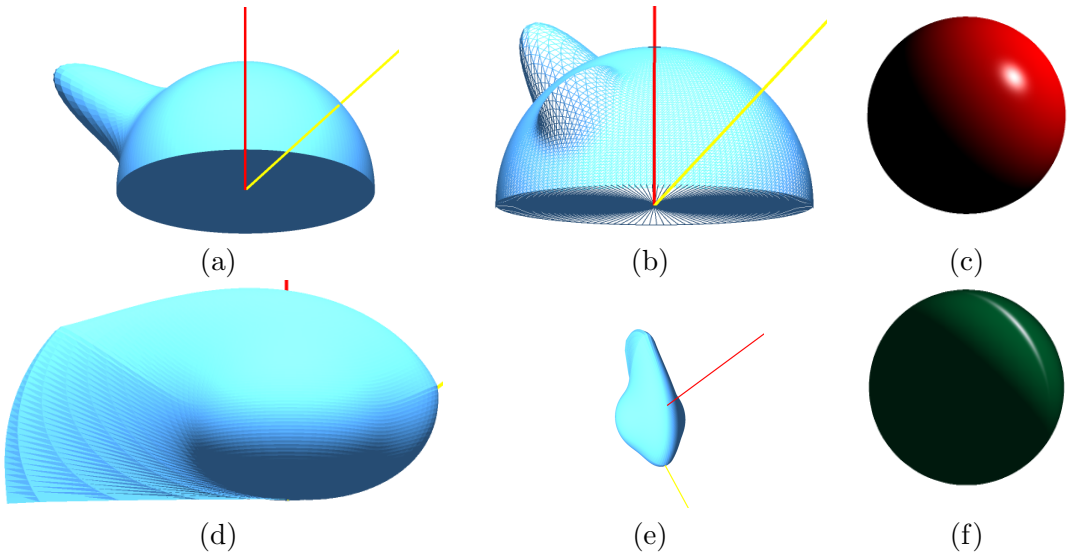


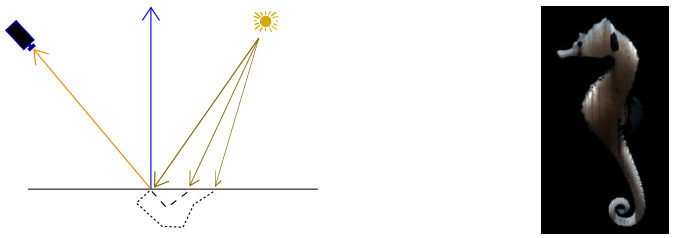
Figure 1.8: Reflectance Functions (BRDF) Example. Red line is the normal, and the yellow the incoming light direction. (a) Phong (isotropic) BRDF. (b) Another view of the Phong BRDF. (c) Phong-lit sphere. (d) Ashikhmin (anisotropic) BRDF. (e) Birds-eye view of Ashikhmin BRDF. Notice that specular lobe is not radially symmetric, but is a function of both θ_o and ϕ_o. (f) Ashikhmin-lit sphere.

a distribution function (as the name says). This means the BRDF only makes sense under an integral. However, since the BRDF falls into the $\rho(x, x', x'')$ term of the rendering equation (Equation 1.1) and is therefore inside an integral, other distribution functions can be integrated against the BRDF. For example, the dirac delta impulse (a distribution), when applied to another distribution, is equivalent to point sampling the other function. Another way of looking at the BRDF is that it is the ratio of outgoing and the incoming light for all pairs of directions. The formal definition of the BRDF, f_r, is

$$f_r(\theta_i, \phi_i, \theta_o, \phi_o) = \frac{L(\omega_o)}{L(\omega_i)\cos\theta_i d\omega_i} \tag{1.3}$$

where $L(\omega)$ is the intensity of light in that direction, $\omega_i = (\theta_i, \phi_i)$ is the incoming light direction in polar coordinates and similarly $\omega_o = (\theta_o, \phi_o)$ is the outgoing light direction in polar coordinates.

The lowest dimensional reflectance function is one-dimensional and describes only the *diffuse* interaction between the incoming and outgoing light directions. The diffuse reflectance only takes into account the angle difference between the surface normal and the



(a) (b)

Figure 1.9: Reflectance Function (BSSRDF) Example. (a) Incoming light can travel through the surface and exit elsewhere. (b) Example of light traveling through the object. The light source is on the other side of the seahorse.

incoming light direction (θ_i). The reflected light is brightest when the light is directly above the surface.

The next most common form of reflectance function involves three dimensions and is known as an isotropic reflectance function. *Isotropic* reflectance functions use the angles between the surface normal and both the incoming and outgoing light directions (that is, θ_i and θ_o, respectively), as well as the rotation angle around the surface normal between the incoming and outgoing light directions, $(\phi_i - \phi_o)$. One example of an isotropic reflectance function is the Phong reflectance function. Another type of isotropic reflectance function is the Blinn-Phong. Isotropic reflectance functions are probably the most common type of reflectance function used in computer graphics.

A well known form of reflectance function uses four dimensions and is known as the anisotropic reflectance function. *Anisotropic* reflectance functions use two axes to generate a frame of reference - the surface normal and an orthogonal axis. The four dimensions are the angles from the surface normal and from the orthogonal axis for both the incoming and outgoing light directions, $(\theta_i, \phi_i, \theta_o, \phi_o)$. Anisotropic reflectance functions are much less common than isotropic reflectance functions, but provide a much more expressive set of representable physical phenomenon. Examples of anisotropic reflectance functions include Ashikhmin's model [5] and Ward's model [103], Lafortune Generalized Phong model [58], and the Banks' model [7]. Brushed aluminum, grooves on a CD, and brushed velvet all exhibit anisotropy in the real world.

Increasing in dimensionality, higher orders of dimensionality allow for much more complex lighting. For example, Jensen's reflectance model for subsurface scattering allowed efficient rendering of "soft" objects such as marble and skin [51]. In addition to the four dimensions in anisotropic reflectance functions, added dimensions take into account the light coming from under the object's surface point (this light arrived by entering the object elsewhere and scattering to the current surface point.) This type of reflectance function is known as the *BSSRDF*, or *Bidirectional Surface Scattering Distribution Function*. Similar to the BRDF, the BSSRDF is not an actual function but a distribution. The reader should refer to the definition of the BRDF given previously for more information on that aspect. Comparing the BRDF to the BSSRDF, the BRDF is a proper subset of the BSSRDF. Whereas the BSSRDF can describe the light interaction between any two rays on a surface, the BRDF can only do so for a single point (both rays must be centered on the same point.) The equation for the BSSRDF, S, is

$$I(x_o, \omega_o) = S(x_i, \omega_i; x_o, \omega_o) dI(x_i, \omega_i) \tag{1.4}$$

The interested reader should refer to [51] for more information.

Any of the above reflectance functions can be modified to include additional effects. For example, by adding increasing the dimensionality, effects such as *translucency* can be added, whereby incoming light can also be described by the light that elsewhere is refracted into the object and leaves at the current surface point. Glass is a common material exhibiting translucency.

1.1.3 Textures

The use of the terms *texture*, *procedural texture*, *image texture*, and *shader* need to be defined for clarity. Since these terms are integral to the understanding of this thesis and have several definitions in the literature, I define and examine these in this section.

The term *texture* refers to a spatially-varying reflectance function. Textures are applied most often to virtual objects and/or simulations of natural phenomenon. During rendering, a texture is queried for the color corresponding to a point, area, or volume. The answer to

such a query can vary in implementation from a value look-up to computation of a complex function. Inaccurate results to the query or inaccurate approximations of the area or volume can lead to aliasing during rendering.

There are several types of textures, and that list depends on how textures are delineated. For example, categorizing textures by application to objects, then textures would fall into surface and volumetric textures. Categorizing textures according to the query implementation could be delineated by procedural textures and data textures (i.e., the query looks up the answer among the texture data.) In this thesis, however, the categories are not separated, but one is a subset of the other. The two categories are image and procedural. The definition of procedural texture is a slight departure from the literature, however, this will increase clarity of the discussion between various types of textures.

Image textures are textures that have a finite level of detail inherent in the pre-computed data used during color queries. Furthermore, an image texture does not usually store reflectance functions, but values to be used to help define the texture's associated reflectance functions. One example of an image texture is, as the name implies, an image that can be applied to an object. Color queries simply look up the corresponding pixel's value. Another example of an image texture is the result of painting a picture onto an object and then unwrapping the paint onto a flat plane for storage. During color queries, the paint color is similarly looked up in the stored image. A more complex example uses multiple image textures as masks for various reflectance functions.

Procedural textures, however, are textures that have no theoretical limit on the level of detail; furthermore, reflectance functions are part of the texture itself. This does not mean that all procedural textures have infinite detail (it does mean, however, that all procedural textures have access to their own reflectance functions.) Rather, a procedural texture's color queries are of the form of an infinitesimal point, and can therefore theoretically have infinite detail. (Although many procedural textures accept color queries of an area, this is a result of the texture author's attempt at anti-aliasing, and is orthogonal to this definition.)

From these definitions, image textures are a proper subset of procedural textures. For example, a procedural texture could look up and return the pixel value in a stored image.

Since the procedural texture is defined on a point, and it could theoretically do anything it wanted with that pixel value, there could still be infinite detail, thereby fulfilling the definition of a procedural texture. In this case, a procedural texture is also an image texture.

In order to simplify the comparison of image textures and procedural textures that lie outside of the image texture subset, the term *shader* will refer to all procedural textures that cannot be included in the image texture set of textures. This is a departure from some of the literature, but allows greater flexibility in the discussion of the procedural textures that are not image textures.

There are numerous methods to anti-alias image textures (due to minification) which work well. These methods, however, do not translate well into the realm of shaders. Chapter 2 will go into more detail of some of the methods for anti-aliasing image textures. The fundamental problem that precludes the use of existing image texture minification anti-aliasing methods for shaders lies within the definition of image textures. These methods have two assumptions that are not necessarily true for most shaders: first, an average reflectance function weight over an area of the texture is known, and second, the associated reflectance functions are diffuse. The second assumption is perhaps somewhat subtle, but poses more of a problem for adapting existing image texture anti-aliasing methods than the first.

First, consider the diffuse reflectance function. The intensity changes with respect to the $cos\theta_i$ term. This is a relatively smooth and slow-changing function. Now consider an instance of this reflectance function on the surface of an object. A single point P_0 with surface normal N_0 uses the diffuse reflectance function for the incoming and outgoing light directions resulting in the color C_0. Moving P a small amount to P_1 changes the normal to N_1. The incoming and outgoing light directions could easily stay the same (if they are sufficiently far enough away). The same reflectance function is used, but a new color is generated C_1. However, since the reflectance function is smooth and changes somewhat slowly, this difference in color is noticeable only when severe changes in N_i exist.

Now consider a simple isotropic reflectance function. Such reflectance functions often

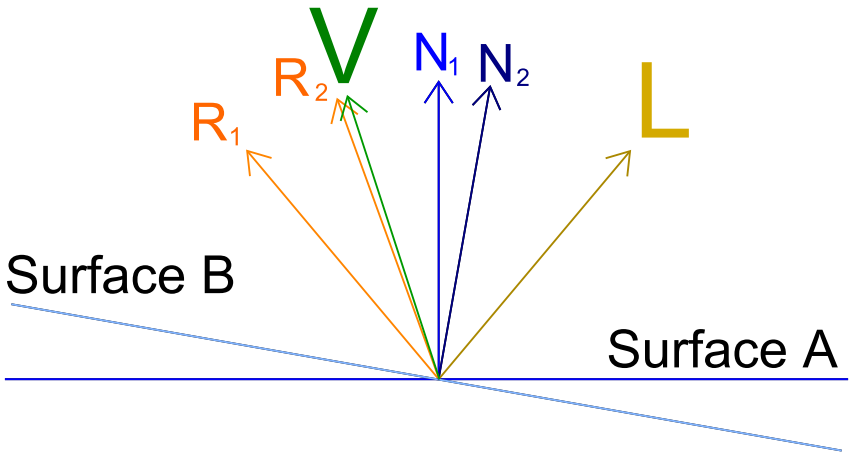


Figure 1.10: Failure of Image Texture Anti-Aliasing Methods Example. Two different normals, N_1 and N_2, are shown. The incoming light direction, L, is in yellow. The viewing vector, V, is in green. The reflection vectors, R_1 and R_2, for the two different normals, N_1 and N_2, are in orange, respectively. Notice how R_2 is very close to where the specular highlight would be. This can cause aliasing if not handled properly.

contain specular components which change rapidly. Moving from N_0 to N_1 could change from a highlighted to a non-highlighted region. Such changes cannot be accounted for in traditional image texture anti-aliasing methods. An example of this is shown in Figure 1.10.

While image textures have many methods that anti-alias minification problems well, there are no equivalent methods for shaders. Furthermore, the only current way to anti-alias some shaders is by hand - that is, they must be coded to give appropriate results during texture minification. More details on anti-aliasing shaders are examined later in this section.

Despite the aliasing disadvantage in shaders, shaders are better suited than image textures for many applications. Image textures are not generated, they are made. Either from a photograph or a digital artist who painted it onto an object, both methods have an image source. Obviously, painting requires effort from a skilled artist, and photographs are not always convenient to acquire. If an object is not flat enough or the image doesn't exist in reality, the capture process is greatly hindered. Furthermore, the capture process will include the current illumination, which is rarely desired. Shaders do not suffer from these problems [4].

Image textures obviously have limited resolution, but shaders do not necessarily have

those, by definition. This means that upon magnification, the image pixels can be seen. Furthermore, limited resolution also applies to large objects. Large objects could require tiling of the image texture, creating often undesirable repeating patterns. Shaders do not suffer from this limitation, either. Furthermore, some textures require details at vastly different scales, and to create an image texture that captures all of that detail could take an artist a lifetime, whereas a shader can be written to do this in less than a year [4].

Often image textures are applicable to only one object. However, shaders can easily be applied to any object and are also easily adjustable for a slightly different looking texture by tuning the shader parameters [4].

Finally, image textures can require a lot of memory, but shaders require only the compiled shader code [4].

On the other hand, there are downfalls to using shaders over image textures. A shader has to be written, and that can take longer to do that an artist to paint the image onto the object. Furthermore, an artist's desire to change one small aspect of the texture is often more easily applied to an image texture than to a shader [4].

The advantages of shaders make them important and commonplace today. Languages for writing shader code is prevalent in computer graphics, including languages such as the RenderMan Shading Language [101] and GPU shading languages such as Cg and HLSL [75]. Easy creation of highly detailed scenes, infinite detail, and tunable parameters make shaders popular today in computer graphics.

However, as stated previously, minification aliasing is a major concern with shaders. Even if there is a lot of detail, if minification aliasing occurs, the benefits of the shader are destroyed with the severe degradation in photo-realism. Therefore, the necessity of anti-aliasing shaders is paramount.

First, anti-aliasing image texture methods were mentioned earlier and that they do not translate well to shaders. This is true because shader color queries are defined at the level of an infinitesimal point with no area. Color queries, however, need the color over an area. Purely from a mathematical standpoint, an infinite number of color queries would be required for an accurate area result. Image textures, however, have pixels as the indivisible

unit, and these pixels are constant in color, allowing for the anti-aliasing image texture methods to work.

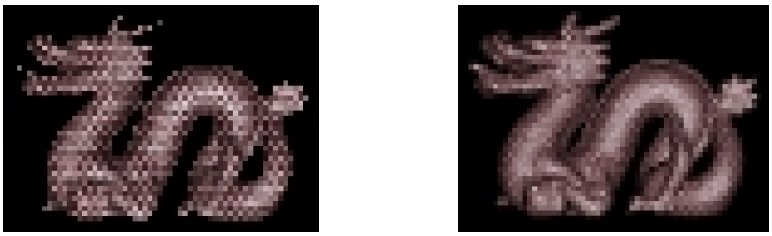
Since traditional anti-aliasing techniques for image textures don't cross over well to shaders, other approaches are required. Currently, anti-aliasing a shader is often difficult and can cost the programmer more time anti-aliasing than it did to create the original shader. The best method, mathematically speaking, to anti-alias a shader by hand is to numerically integrate the shader over the texture footprint. However, this is difficult or impossible most of the time [4].

The cost of calculating the weighted average of colors that fall within the texture footprint can be prohibitive for a shader, requiring many samples due to the arbitrary number of colors that can be mapped to a single texture footprint [75].

Automatic anti-aliasing shaders could be accomplished by rendering the texture at a very high resolution to use as an image texture [19]. Unfortunately, many of the benefits of shaders are lost in this conversion. Finite detail and limited reflectance functions makes this option questionable. This is similar to our approach, but we maintain both the infinite detail as well as the reflectance functions during the conversion.

A different method of automatic anti-aliasing shaders is super-sampling during rendering. In this case, though, unnecessary samples are required elsewhere in the scene, unless the shader itself is programmed to super-sample the texture footprint. Even then, when the texture footprint is large, a prohibitively high number of samples might be required to arrive at a good estimate. Moreover, none of this computation is re-used in later frames.

Another possibility is to use multiple frames to blur places that are suffering from aliasing, however this presents several problems. There are numerous difficulties in identifying the correct surface features, including scene segmentation, blob tracking, and having portions of the object leave and enter the frame sequence. For a description for post-processing for motion blur, see Brostow and Essa [16]. Once the surface has been identified, it is not obvious as to how to automatically determine when aliasing is occurring. Furthermore, the animation could easily suffer from too little or too much blurring, depending on the accuracy of aliasing detection. The other problem is that post-processing of the frames does not



(a) (b)

Figure 1.11: Comparison of Automatic Super-Sampling vs. PRM Example. Zoom of Distant dragon. (a) 16 samples per pixel. (b) Procedural Reduction Map with 1 sample per pixel.

guarantee high visual fidelity to the actual answer. If each frame is a poor reconstruction of the actual scene, then post-processing the frames will not help.

1.2 Procedural Reduction Maps

The goal of chapter 3 is to minimize and/or eliminate minification aliasing from most procedural textures while maintaining high visual fidelity to the original texture. Working with a minimal set of assumptions, I introduce the *procedural reduction map* to achieve this goal. The procedural reduction map combines information about both reflectance functions and distributions of surface normals into texels for anti-aliased rendering of procedural textures. The method includes automatic generation of the texels off-line to minimize run-time rendering costs. A comparison example of both current automatic methods and the procedural reduction map are shown in Figure 1.11.

This method minimizes and/or eliminates the minification aliasing problem found in procedural textures while maintaining high visual fidelity to the original procedural texture. Other published methods in the graphics literature either severely limit the type of procedural textures created [74] or greatly degrades the visual fidelity to the original texture [76]. Furthermore, with procedural reduction maps, the magnification and reflectance function benefits are maintained, surpassing traditional image texture techniques such as surveyed in [44].

To achieve this goal, the procedural reduction map employs a pyramid of texels, similar to MIP-Maps. As stated earlier, texels contain both normal distributions and reflectance function information. The procedural reduction map algorithm samples both the object's

surface and the procedural texture along the surface, creating texels from that sampling that contain reflectance function and surface normal distribution information. The procedural reduction maps partially consists of several points on the object's surface whose corresponding texture reflectance functions can fully reproduce all sampled reflectance functions on the surface. These few basis reflectance functions are referred to in each texel with weights to reproduce the reflectance function for that texel. The research presented here requires a surface-to-UV mapping that preserves area ratios. The creation of the UV mapping of the objects will not be covered in this thesis, but details of this can be found in the work of Zhang et al [110].

1.3 NBRDFs

The second contribution of this thesis, the *Normal-centric Bidirectional Reflectance Distribution Function*, or *NBRDF*, continues the idea of anti-aliasing procedural textures, but concentrates on the representation of the reflectance functions in the procedural reduction map texel. Chapter 5 explains the need for better integration of BRDFs and the solution: NBRDFs. NBRDFs sample a reflectance function and store the sampled function relative to normal distributions. This method of BRDF storage answers the problem of BRDF integration over normal distributions. Fast access to the BRDF normal integration at run-time allows for better approximations to the original signal, and thus less aliasing. NBRDFs are capable of representing isotropic BRDFs of varying forms including multi-specular-lobed BRDFs.

However, the benefit of NBRDFs is not limited to anti-aliasing. Through the use procedural reduction maps with NBRDFs, graphics hardware implementation becomes possible. Furthermore, procedural textures that had too many lines of code to put on graphics hardware suddenly become feasible, thereby opening new possibilities for rendering high quality textures in real-time that could once only be done off-line. An example of NBRDF use in a procedural reduction map is shown in Figure 1.12.

Figure 1.12: NBRDF Example. A multi-lobed Phong variant.

1.4 GPU Implementation

Chapter 6 explains implementation details and algorithmic changes necessary for NBRDFs to be rendered with modern day graphics hardware. The *Graphics Processing Unit*, or *GPU*, combines the parallel computing with the parallel nature of the rasterization process. Other published research demonstrates BRDFs on the GPU, such as McCool et al.'s work [71], however, these do not make an effort at anti-aliasing or handling multiple reflectance functions simultaneously.

Although nothing new is presented here, chapter 6 describes the theoretical challenges of using procedural reduction maps and NBRDFs on GPUs. Some of these challenges include limited number of texture look-ups, limited texture resolution, limited texture size, and limited number of rendering instructions.

The result of implementing procedural reduction maps and NBRDF on GPUs is that complex procedural textures become viewable on graphics hardware, both anti-aliased, and with high visual fidelity to the original texture. Furthermore, if procedural reduction maps were implemented at the hardware level and if a procedural texture was implementable on the GPU, then the magnification benefits of procedural texture could be maintained. A snapshot of a GPU-rendered procedural reduction map with NBRDFs is shown in Figure 1.13.

Figure 1.13: GPU Example. A bump mapped bunny on graphics hardware.

1.5 Contribution Overview

There are four major contributions of this thesis, three of which concern the procedural reduction map, and the fourth the NBRDF:

- Using a procedural reduction map for anti-aliased procedural textures is automatic, given a procedural texture on an object using a minimal set of assumptions. Previous methods for anti-aliasing specify the way the texture is written or are not automatic.
- The procedural reduction map minimizes the aliasing artifacts of procedural textures due to minification without losing the benefit of magnification details.
- For complex textures, the procedural reduction map more closely resembles the original texture than a simple image texture map by using reflectance functions at each texel in the pyramid.
- The NBRDF allows for fast run-time integration of general isotropic BRDFs, giving higher quality anti-aliasing results and the new capability of moving procedural reduction maps onto GPUs.

The are several assumptions about the model and the procedural texture that are necessary for procedural reduction maps to work. These are included here, but discussed in more detail later. Some of these assumptions can be relaxed given a little input from the texture author and will be discussed more in chapter 3.

- Procedural texture generally obeys the properties of light.

- Beyond shadows and light sampling, procedural textures cannot sample the scene.
- Procedural textures must use reflectance functions no more complex than isotropic reflectance functions. (Non-refracting transparency is allowed.)
- Procedural reduction maps will not identify different reflectance functions when such functions are different only in specular lobes outside the reflected vector (from the incoming light).
- Procedural textures can be applied only to non-deformable models and cannot use displacement mapping.
- The procedural texture must be time-invariant.

The assumptions for NBRDFs is listed below, but will be more fully developed in chapter 5.

- The input BRDF must be isotropic.
- The input BRDF cannot be translucent in any way.

The next chapter will present work that is both related and important for understanding the background of the material presented in this thesis.

CHAPTER II

PREVIOUS WORK

There are several sections in this chapter that explain the history of various aspects to anti-aliasing procedural textures. The following sections explain them. First, reflectance functions and textures are discussed. The third section discusses anti-aliasing textures, while the last two sections discuss normal distributions and real-world sampling.

2.1 Reflectance Functions

Reflectance functions are at the heart of all realistic procedural textures. Their importance is independent of procedural textures and even pre-dates them.

There have been numerous models proposed for describing reflectance functions. The first reflectance function that had both diffuse and specular components linearly combined was proposed by Phong [86]. Later Fresnel effects were added by Blinn [10], based on Torrance and Sparrow's research [100].

However, these models were greatly simplified from reality. Cook and Torrance introduced a reflectance model that could handle rough surfaces and were based much more on optics [24]. From this time on, more reflectance models were presented to better resemble reality. Kajiya introduced another important reflectance function model that had anisotropic properties [52]. Another choice that later became popular for anisotropic models was introduced by Poulin and Fournier that could also describe refraction [87].

Photo-realistic rendering took a large step forward when Cook et al. introduced the reflectance equation [28] and Kajiya introduced the rendering equation [53]. Several concepts were solidified into the rendering equation. This equation described the entire goal of photo-realism: the complete simulation of proper light interaction within a virtual scene [53]. Meanwhile, Cook et al.'s reflectance equation described the distribution of light that is reflected on an object's surface. Cook mentioned that the reflectance equation may be too complex to compute analytically, but that this more accurately describes an object's surface

appearance [28].

So far, though, these reflectance function models fell under two categories: they were either empirical or theoretical. Empirical versions started with a model that might or might not later fit the parameters to real data. Theoretical on the other hand starts with sound physical theory and then does parameter matching. However, Ward started with real data and made a relatively simple anisotropic model from that data [103]. This started a new trend of types of introduced reflectance models that started with real-world data. An example of this is the new reflectance model introduced by Matusik et al. The model was not analytical, but rather a dense set of measurements that are then dimensionally reduced for rapid rendering [68].

More advanced models began to appear. Westin et al. introduced a third layer to the BRDF, the micro-scale, allowing for better scattering analysis [104]. Schlick introduced an inexpensive BRDF model that also handled anisotropy, but also included more realistic combinations of Lambert and Fresnel surfaces as well as heterogeneous materials [94]. Lafortune et al. introduced a non-linear model of BRDFs that expressed a wider range of materials with more realism [58]. Ashikhmin and Shirley use the Phong specular lobe, but include factors to make the model physically plausible. Their model is expressive and implementable on graphics hardware [5]. Ashikhmin et al. also created a BRDF model that was physically plausible, but handled general microfacet distributions [6].

However, BRDF models were not the only thing being advanced in the area of reflectance functions. Subsurface scattering also appeared in literature. Blinn took the first step in subsurface scattering when he described methods for handling the rendering of clouds and other diffuse surfaces [12]. Hanrahan and Krueger simplified his model to be useful in general surface reflectance functions [40]. However, their model only did single scattering interactions. Jensen et al. extended this model to include multiple light scattering interactions in the fast BSSRDF [51].

A different approach to representing reflectance functions was basis summation techniques. Using a sum of simpler functions, a BRDF is the resulting sum of those simpler basis functions. For example, Phong lobes can be used as basis functions [102].

With the rise of programmable graphics hardware, many BRDF models were reworked into new representations so that they could run on the new hardware. One such type of representation was spherical wavelets introduced by Schröder and Sweldens [95]. Later, Ramamoorthi and Hanrahan [88] used spherical harmonics to describe diffuse illumination with global effects. Other methods followed the use of spherical harmonics. For example, Ramamoorthi and Hanrahan again used spherical harmonics to create a spherical harmonic reflection map that could include isotropic BRDFs [89]. Kautz and Seidel introduced a method for handling bump maps with anisotropic BRDFs on hardware, using a form of factorization [57]. Gershbein and Hanrahan factorized BRDFs for fast rendering through the use of graphics hardware [36].

Other representations grew as well, all centered on the graphics hardware. Specifically, factored representations of BRDFs became important, where the BRDF is represented by a series of textures whose composition creates an approximation to the BRDF.

Heidrich and Seidel analytically separated the Banks anisotropic model for factorized rendering [45]. Kautz and McCool then extended this idea to generalized BRDFs [55]. Heidrich and Seidel combined factorization with environment maps to achieve shadowing and Fresnel factors [46]. McCool et al. produced a homomorphic factorization that had no negative terms and better relative error, allowing for better hardware implementation [71]. Latta and Kolb generalize the homomorphic factorization to include the global illumination for isotropic BRDFs [59].

Factorization was also proving useful in reflectance function integration. Lawrence et al. presented a factored representation in order to provide better sampling of the distribution during rendering [61].

2.2 Image and Procedural Textures

In 1974, Catmull introduced the first textured object to computer graphics [20], and Blinn extended it to include bump maps, reflections, and roughness [11, 13, 14]. However, the technique of writing code to define a texture truly blossomed when Cook introduced the concept of Shade Trees [25], followed by the next year's work by Perlin, solid noise and the

Pixel Stream Editing (PSE) language [84], along with solid textures (simultaneously with Peachey [81]).

Shade Trees changed the manner in which surface appearance was put together. The algorithm separates different aspects into different trees, where any tree can be grafted onto numerous other trees. Complex surfaces became much easier to represent through identification of underlying elements (such as color, lighting) that could be combined through operators such as dot products, interpolation, etc. Through the use of Shade Trees, practical generation of overall realism of surfaces improved greatly [25]. Later, Abram and Whitted described an implementation of the Shade Tree algorithm, extending its use [2].

Perlin extended this idea with the PSE language [84], allowing for interactive generation of new procedural textures. Furthermore, the operations were performed at the pixel level, effectively the first fragment shader.

Perlin also introduced band-limited solid noise [84]. Solid noise allowed easy creation of solid textures through pseudo-random noise that obeyed certain desirable characteristics (such as continuity in $\mathbb{R}^3$). Convincing procedural textures of wood, clouds and marble were easily produced through this new method.

Band-limited noise can be considered a texture basis function, which defines a scalar value over all of space, $\mathbb{R}^3$. While useful in generating many different types of appearances, it could not easily generate all such effects. Worley introduced the cellular basis texture function that created points and scattered them in space. The scalar function would relate how close a given point is to the scattered points [108].

Image textures themselves, though, were becoming more complex as well. Oliveira et al. introduced relief texture mapping that demonstrated 3-D surface details and view motion parallax [80]. Textures in general require parameterization on the surface. Benson and Davis introduced octree textures that require no such parameterization and can therefore more easily be used on traditionally difficult surfaces, such as implicit surfaces. This is accomplished by storing information in an octree format [9]. Another representation is the shell texture function algorithm proposed by Chen et al. They describe a texture as a 3-D layer that contains the texture information, allowing for meso-structures and volumetric

variations in the surface [21].

However, Perlin's PSE language was not the last language to impact procedural textures. Procedural textures became even more prevalent in computer graphics in the 1990's with the introduction of shading languages, beginning with Hanrahan and Lawson's work. They incorporated ideas from both Cook's and Perlin's previous works to create a high level language that used abstract shading models, allowing for more expressiveness [41]. The interface was according to the RenderMan interface specification [1], which had recently been explored through Upstill's book [101]. The new shading language also provided for more compact shaders [41].

More shading languages appeared, the most notable of which was the PixelFlow Shading System, by Olano and Lastra. Real-time image generation of procedural textures through a graphics multi-computer and its own (but very similar to RenderMan) shading language [77].

In recent years, high level programming languages have appeared for graphics hardware, including, but not limited to, Brook [17], Cg for nVidia cards [67], and DirectX [15].

2.2.1 BRDFs in Procedural Textures

Textures and BRDFs have often been closely tied to each other. Blinn and Newell introduced environment maps that tied global illumination to the object's appearance [13]. Heidrich and Seidel approximated real-time glossy reflectance using a pre-blurred environment map with the Phong model [47]. Later, Kautz and McCool extended the idea to include arbitrary isotropic BRDFs [56].

Dana et al. introduced a new texture representation called the BTF (bidirectional texture function) that represents different reflectance functions over the surface of a real-world object, as well as providing a database of reflectance measurements. The BTF describes the dependence of surface texture upon incoming and outgoing light directions. Although a little existed in the literature previously, the BTF was the first to focus on this dependence [31]. Liu et al. extended this work to decrease the required sampling of the six dimensional function [64].

2.3 Anti-aliasing

The history of anti-aliasing in photo-realistic rendering is long and covers the various sources of aliasing mentioned in the previous chapter. The next few sections will stick mostly to minification aliasing.

2.3.1 General Anti-Aliasing

The first method for anti-aliasing worked with any renderer - rasterization and ray tracing included. Crow introduced the concept of super-sampling to the computer graphics world from signal theory. This didn't solve the problem of anti-aliasing, but it did move the frequency at which it occurs higher. Furthermore, he also introduced convolution filters to blur images to decrease aliasing [29].

Dippé and Wold introduced stochastic sampling to computer graphics [32]. Cook also independently introduced jittering [26]. Stochastic sampling converts aliasing energy to noise energy, which is less bothersome to the human visual system. It can be used for most sources of aliasing, including silhouette, minification, and temporal anti-aliasing. Mitchell extended this idea to decrease the number of such stochastic sampling [72]. More generally, Mitchell also described how to generate filters for rendering purposes [73]. These methods mostly apply to ray tracing, where it is easy to change the sampling pattern.

Cabral et al. presented a method for anti-aliasing the Torrance-Sparrow reflectance function by estimating the integration of the BRDF by the global illumination by using spherical harmonics [18]. McAllister et al. created a pyramid, where each texel had the coefficients for the Lafortune BRDF [69]. However, this doesn't truly address reflectance function integration, but rather linearly weighted combinations of function parameters. Since many of these are nonlinear, the linear combination is not enough. Tan et al. introduced the method of multiresolution reflectance filtering for general integration of reflectance functions. Their method does a much better job of performing general reflectance function integration. They treat the input to the reflectance function as a Gaussian mixture model of normal distributions [98]. Their method closely resembles the NBRDF, but only handles Lambertian and Cook-Torrance models. Furthermore, their texel representation does not represent a

Gaussian distribution of normals.

Shirman and Abi-Ezzi introduced the cone of normals technique to increase the speeds of different aspects of the rendering process, including patch tessellation and culling, as well as possible silhouette detection. Their method described patches with a *floating cone of normals* that could be used for fast comparison [97].

2.3.2 Image Texture Anti-Aliasing

Image texture anti-aliasing has received a lot of attention in the computer graphics literature due to its prominence. Heckbert performed an excellent and extensive survey of image anti-aliasing [44].

Williams produced the now famous MIP-Map algorithm where for color texels within a pyramid. His proposal also included the better trilinear interpolation scheme [106], an improvement upon Dungan et al.'s scheme for pyramids [105]. Amanatides extended the use of MIP-Maps in ray tracing by using cones to ray trace and to give an estimate of the texture footprint as an indicator of what level to use in the pyramid [3].

Other techniques exist to anti-alias image textures, one of which is *repeated integration filtering*. The technique first appeared when Crow introduced summed area tables, which allowed arbitrary rectangles to be filtered in constant time. The method stored high precision values of summed components of the image texture [30]. Both Perlin and Ferrari et al. independently generalized summed area tables using a technique called repeated integration [34, 43, 83]. This technique allows for elliptical filtering of the image texture.

Another method was introduced by Schilling et al. called footprint assembly [93]. This performed filtering on the original image texture for better, but often more costly texture footprint filtering. In general, texture footprint estimation is a key aspect to anti-aliasing both image and procedural textures. It is a difficult aspect to get right, but can make all the difference in the final rendered result [33]. Igehy introduced a method for texture footprint estimation by tracing ray differentials [50]. Many textures use some type of texture differential to determine an estimate of the texture footprint [33].

Extensions to these methods were made, such as anisotropic filtering for the texture

footprint. Summed-area tables, as mentioned before, was one of the first anisotropic filtering techniques for image anti-aliasing [30]. However, early anisotropic filtering still failed when the anisotropy was diagonal in texture space. Olano et al. introduced a vertex-based anisotropic anti-aliasing scheme that could be run on programmable graphics hardware that better handles such cases by using multiple texture accesses [78].

As stated before, image texture magnification is not technically aliasing. However, we mention it here again to state that it remains an issue today. For example, recently Sen introduced silhouette maps to better deal with this problem [96].

2.3.3 Procedural Texture Anti-Aliasing

The open-ended nature of procedural textures provided power and flexibility, however, the same flexibility increased the difficulty to automatically anti-alias the same texture and has therefore been attempted by few researchers.

There are several programmer-oriented approaches to anti-aliasing procedural textures, and an overview of many of them can be found in [33]. One of these techniques is clamping, which was described by [74]. Clamping effectively removes aliasing, but is appropriate only for spectrally synthesized textures. Analytically pre-filtering the texture is an effective means to remove edge aliasing and can be used as a stepping stone for minification anti-aliasing, but generally it is difficult except for simple filters [33]. Numerically integrating the texture is obviously the best choice, but is usually the least tractable method – sometimes even impossible [4, 43].

One of the problems with manual techniques is that often the programmer spends as much time anti-aliasing as creating the texture itself. A more computer-centric method is to simply sample with higher frequency and to jitter the samples. This, however, will not remove aliasing, but merely change the frequency at which aliasing occurs [33, 75]. Heidrich et al. introduced an excellent method of determining a tight bounding error on procedural texture sampling. The bounding error was calculated by using affine arithmetic on the shader parameters [47].

A lot of the complexity of the anti-aliasing problem arises from the use of the original

Perlin noise function. Some of these problems were solved by Perlin himself [85]. This work did not, however, address the fundamental problem in anti-aliasing noise, which is that it is not strictly band-limited. Wavelet noise addressed this issue so that noise became band-limited [27]. This does solve a subset of the problem of anti-aliasing procedural textures. Unfortunately it does not help in the anti-aliasing of high frequencies that arises when the noise values are subsequently modified, especially by discontinuous functions due to conditional branching, nor in issues of reflectance function integration and weighting or cellular texture functions.

Hart et al. demonstrated a new model for solid texturing, yielding an automatically derived box filter width from the texturing parameters. Their method used a first order approximation of the color index variance [42]. While a good step in automatic anti-aliasing, color index variance within a pixel is not always a good indicator, and for many procedural textures, there is no color index.

2.3.4 Procedural Texture Simplification

In addition to having arbitrary complexity, procedural textures can be of arbitrary length as well. If the length of the texture is too long, it won't fit onto graphics hardware. To handle the combined issues of anti-aliasing and length, many techniques have been introduced to simplify the texture itself.

Originally, creating shaders for multiple levels of detail was a manual process. Goldman manually created a fur texture at multiple levels of detail [37], which is similar to Apodaca and Gritz's method of blending between two or more textures when one of them begins to alias [4]. Olano and Kuehne created a set of shader building blocks that inherently had levels of detail. When procedural textures used those building blocks, they also inherited the levels of detail [75]. When only a few parameters were expected to change, Guenter et al. proposed the *specialized shaders* algorithm [39]. Olano et al. introduced a completely automatic level of detail system for shaders that tried to minimize texture accesses for hardware implementation [76]. This work, like all level of detail shaders, have problems with popping when going between levels. This was also true of Pellacini's work that applied

simplification rules to the shader code and then chose the one that minimized the error [82].

A concern with all of the simplification methods is the lack of visual fidelity to the original procedural texture. Such change in appearance is often unsatisfactory.

Ma et al. introduced a representation that handled multiple levels of detail for rendering BTFs using a Laplacian pyramid [65]. Although their method resembles procedural reduction maps, it in fact has several large differences. First, its input is real world reflectance function data, not synthetic. Second, while the data is linearly weighted, the reflectance function is not anti-aliased. Third, approximations to the light field are made for pre-computation purposes.

Carr and Hart introduced a slightly different method, but similar to ours. They proposed mapping solid textures onto a surface that had a MIP-Mappable texture atlas applied to it [19]. Their main focus was on the texture atlas that would be amenable to trilinear interpolation after a pyramid was created from it. This approach grew out of a similar method for storing surface attributes for scanned meshes [22, 90]. Cohen et al. introduced a technique for preserving appearance during model simplification, but it assumed image textures were applied to the object and nothing more complex [23]. Sander et al. also tried to maintain texture coordinates during mesh simplification, but also suffers from the same downfall [92].

Similar to the previous methods, many researchers have tried to have level-of-detail (LOD) meshes for faster rendering. One such method is the Progressive Meshes method by Hoppe [49]. Another geometry-based solution to smooth a mesh is Taubin's algorithm [99]. It may be that these algorithms can help with the problem of highlight aliasing. These algorithms assume that the geometry already contains all of the normal information. However, many procedural textures use bump maps and these are inherently not in the geometry. Furthermore, as will be shown in chapter 5, even if the smoothing was done perfectly, the underlying problem of single sample integration sampling of the reflectance function is typically not enough for good anti-aliasing. A final problem is the complexity involved in changing smoothly between various polygon models during animation.

2.4 Normal Distributions and Bump Maps

During minification of bump maps, it was pointed out by Fournier that traditional methods of anti-aliasing them (i.e., by averaging the normals) would not work and proposed the idea of creating a pyramid of Phong-like distribution of normals [35]. Olano continued this work to include Gaussian distributions of normals that allows for easy anti-aliasing and cheap surface evaluation [79].

Becker and Max introduced a different method for handling bump rendering by blending between the three types of textured bumps: displacement mapping (when the object is relatively close), bump maps (intermediate), and BRDF (for distant) [8].

Kautz et al. furthered the goal of bump map anti-aliasing by introducing a real-time bump map synthesis algorithm. The technique allowed for an infinite zoom into the surface with consistent detail being created on the fly [54]. However, their aim was not the anti-aliasing of an existing bump map, but rather the synthesis of a new one. Therefore, if applied to an existing model, visual fidelity would decrease.

2.5 Real-World Sampling

The acquisition of surface shape from real world objects is a major topic in computer vision and the literature bears that out. Although an entire chapter could be spent describing these techniques, only a few will be mentioned here that are somewhat related to this thesis.

Although shape acquisition is not truly a goal in this thesis, the detection of normal perturbations is, and as such similar methods are noteworthy. Specifically, Witkin describes a method for surface extraction based on a simple lighting scheme of a textured object [107]. Rushmeier et al. designed a scheme to capture fine scale geometry, or bump maps, from objects by placing the lighting in different directions with only a few assumptions about the reflectance function [91]. Zickler et al. used reciprocity to determine bumps in the surface [111]. Hertzmann and Seitz introduced general shape reconstruction with general, varying BRDFs [48]. Their work, however, assumed that there was a previously known object that was similar to the original in reflectance and shape.

Malzbender et al. present a texture that captures a greater part of the light field of a

texture, giving greater realism. They use biquadratic polynomials to represent the different lighting schemes of the texture, with the data captured from real world photographs [66].

Lawrence et al. introduced Inverse Shade Trees to handle editing of materials in real time. They reduced multi-gigabyte data through matrix factorization to achieve a small set of *leaves* in the shade tree representation [60]. This work is similar to this thesis, but varies in several specific ways. First, the Inverse Shade Tree manipulates real world data to achieve the shade trees. Second, the goal is to edit such leaves. Third, anti-aliasing is not a goal of their work and as it stands, cannot handle minification anti-aliasing.

CHAPTER III

MINIFICATION ANTI-ALIASING IN PROCEDURAL TEXTURES

The key to my approach for anti-aliasing procedural textures is the procedural reduction map. This chapter provides a detailed look at the creation and rendering of procedural reduction maps, including concrete examples of an implementation of the procedural reduction map for a ray tracer.[1]

While this chapter deals primarily with describing what a procedural reduction map is, chapter 4 examines the difficulties, assumptions, and decisions from this chapter. Therefore, such discussions will be postponed until then.

The rest of the chapter starts with an overview of the entire procedural reduction map algorithm, from creation to rendering. Next, there is an in-depth look at the creation process. From there, the rendering is examined, followed by results and conclusions.

3.1 Algorithm Summary

The intention here is to give an overview of the procedural reduction map, starting with a few of the most important assumptions about the types of procedural textures that can be used as well as the data structure. This will make the exposition easier to follow.

First, we assume that the input procedural texture uses isotropic reflectance functions whose unique specular components are along the reflection direction. Second, the texture is neither time-varying nor samples the scene beyond lights and shadowing. Finally, the procedural texture attempts to be photo-realistic, that is, generally obeys the physical properties of light. An in-depth discussion of these assumptions and others will be examined in the next chapter.

[1] See chapter 6 for details on a procedural reduction map implementation for a rasterization renderer.

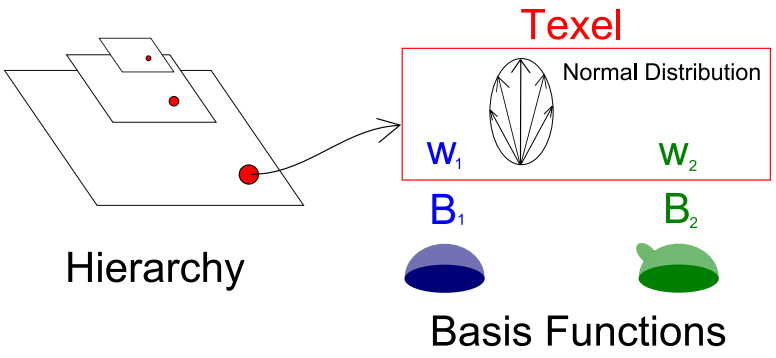


Figure 3.1: Overview of Procedural Reduction Map Data Structure. The procedural reduction map is made up of a pyramid of texels and a set of basis reflectance functions. Each texel contains weights for each basis reflectance functions and a distribution of surface normals.

3.1.1 Data Structure

Procedural reduction maps are composed of two parts: the procedural reduction map hierarchy of *texels*, and a set of *basis reflection functions* as shown in Fig. 3.1.

Each texel in the hierarchy has two parts: the *normal distribution map* and the *basis weights*. The normal distribution map is a Gaussian distribution of normals representing the curvature of the object at the corresponding texel. The basis weights describe the influence the basis reflectance functions have on the texel.

The basis reflectance functions are a small set of reflectance functions that, when linearly combined, describe any of the reflectance functions found on the sampled surface.

3.1.2 Creation

From a high-level viewpoint, the algorithm performs the following steps.

1. Rasterize the polygons of the model into the texture atlas to produce the corresponding sample points on the object's surface.

2. Analyze the reflectance functions at each of these sample points and determine the basis reflectance functions.

3. Re-visit the surface sample points to find the basis weights for each of the basis reflectance functions.

4. Aggregate the texels' weights up the pyramid.

5. Aggregate the normals from the bottom level of the pyramid upwards into distributions of normals for each texel.

The algorithm begins with the user specifying the procedural texture, the (u, v) parameterized object, and the desired size of the procedural reduction map.

The first step creates samples on the object's surface corresponding with the texels in (u, v) space. The desired size of the procedural reduction map determines the number of the texels. From these samples, we can analyze the procedural texture's reflectance functions.

The next step samples the reflectance function at each texel and stores the result as a vector for each texel. Performing matrix factorization on the set of all texel samples determines the basis reflectance function.

Using the basis reflectance functions and their corresponding vectors, the next step projects each texel's vector into the space defined by the basis reflectance function vectors. A texel's vector projection becomes the basis weights for the basis reflectance functions.

In the next stage, we analyze the procedural texture at each sample point to determine the surface normal that the procedural texture uses for lighting calculations. This normal is stored in the corresponding texel.

In the final stage, both basis weights and normal distributions are aggregated up the pyramid of texels until all texels are filled.

3.1.3 Rendering

During rendering, the input to a procedural reduction map is a (u, v) coordinate and an estimate of the texture footprint for a pixel. If the texture footprint is small enough, then the original procedural textures is used. If the footprint size is in-between, a blending occurs between the procedural reduction map's answer and the original procedural texture. Otherwise, the procedural reduction map returns its estimate of the anti-aliased version of the procedural texture.

Given that the texture footprint is large enough, the location within the pyramid is computed from the footprint size and the (u, v) coordinates, similar to MIP-Maps. Using

trilinear interpolation, the next step computes a new set of basis weights, and then renders each basis reflectance function weighted accordingly to give the final answer. Under certain circumstances, screen-space anti-aliasing is performed to compensate for imperfect reflectance function integration.

3.2 Creating Procedural Reduction Maps

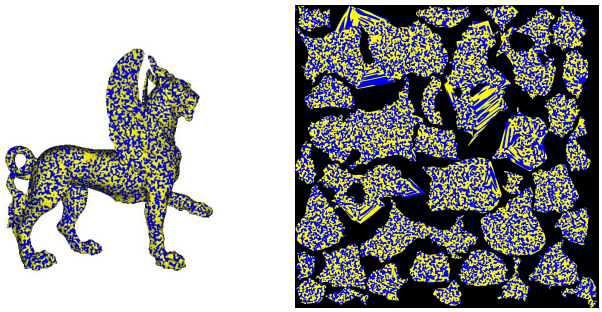
Here, the details of creating the procedural reduction map will be examined. The first detail is the inputs to the creation process. The user specifies the procedural texture, the (u, v) parameterized object, and the size of the pyramid of texels. (The first two inputs are possibly independent. The procedural texture might or might not use (u, v) coordinates in its processing. Either way does not matter for the purposes of this algorithm.)

3.2.1 Texel/Sample Point Correspondence

In order to point sample the object, we need to create a correspondence between the 2D texels and areas on the 3D surface. For simple objects, such as spheres and cylinders, analytic mapping can be used. For more complex items, such as meshes, we use the texture atlas approach developed by Sander et al [92].

The (u, v) parameterized object should have one or more texture patches that have been amalgamated into one texture atlas. Figure 3.2a demonstrates the atlas and its patches corresponding to the textured object shown in Figure 3.2b. There is no assumption about the method used to create the atlas, however, atlases whose (u, v) patches all have the same proportions area-wise to the surface area they represent will increase the effectiveness of the procedural reduction map (similar to MIP-Maps). Although many other methods exist, for the examples used in this chapter, Zhang et al.'s automatic method for texture atlas creation was used [110].

The (u, v) map is then broken into $T_N = D * D$ texels specified by the procedural reduction map base dimension size, D. Each texel relates to an area on the surface of the object. To determine the correspondence, we rasterize the *model elements*, the basic building block of the object. A model element varies based on the type of object. For a triangle mesh, the model element is a triangle. For an object made of Bezier patches, it is



(a) (b)

Figure 3.2: An Example of a Texture Atlas. (a) A procedurally textured mesh is unfolded into (b) patches in (u, v)-space.

a Bezier patch. The (u, v) coordinates of each model element are rasterized to create a set of surface points, S_{sp}, that each have (u, v) coordinates.

The rasterization does not correspond to one sample per texel, but rather is super-sampled at four times per texel.[2] Later, this allows for blending between the procedural reduction map and the original procedural texture when minification no longer occurs. Furthermore, super-sampling S_{sp} allows for the identification of texels that have more detail and thus require further sampling to get an accurate reconstruction.

At this point, we have a list of texels $(T_1, T_2, \ldots, T_N)$, each covering a portion of the object's surface area, along with a list of corresponding sample points on the object (S_{sp}). With these texels and their associated area, we can now analyze the reflectance functions at the corresponding location on the surface of the procedurally textured object.

3.2.2 Basis Reflectance Functions

A fundamental aspect of the procedural reduction map approach is that most procedural textures can be approximated well by a weighted sum of a few *basis reflectance functions*. The goal of this section is to determine this set of basis reflectance functions, S_B. The number of basis reflectance functions should be small (around three to six) in almost all cases. There could easily be over a million of functions to choose from (if the resolution of the pyramid is 1024×1024), therefore we must identify the best functions. The source of

[2]This can automatically be increased if the resulting samples are too different. This is discussed in detail in section 3.2.2.4.

these functions is the sampled reflectance function at each texel's corresponding point on the surface.

Many procedural textures could be written in such a way as to provide the reflectance function at a specific point, which would make basis identification easy. However, it is not necessarily straightforward in all textures to do so. Furthermore, pre-existing textures will likely not have been written in such a way. Therefore, for these cases, we wish to recognize the basis reflectance functions in an entirely automatic way.

To identify these functions, something must be known about the functions themselves. Since this approach is automatic, nothing is known about the functions and therefore the functions must be sampled. Section 3.2.2.2 explains the details of such sampling and won't be discussed here except to note that the sampling will be uniform across each texel. The result of each texel's (t_i) sampling will be a vector (v_i) of the sampling of that texel's reflectance function, so that if two texels have the same reflectance function, they will also have equal vectors. Formally, if $R(t_i)$ is the reflectance function at texel t_i, then

$$\forall i, j | [R(t_i) = R(t_j)] \Leftrightarrow [v_i = v_j] \tag{3.1}$$

The span of the vectors v_i results in a sub-space of vectors. For interesting (i.e., non-zero) vectors, such a sub-space can be spanned by an infinite number of different vectors. Therefore, the number of different possibilities for basis reflectance functions can also be infinite.

Despite this, there are some preferences regarding the selection of the basis reflectance functions. First, and foremost, the vectors representing the final basis reflectance functions should exist among v_i.[3] Second, we require a small number of basis reflectance functions for both speed and storage concerns. Finally, the basis vectors should be as orthogonal as possible to avoid numerical inaccuracy problems during processing.

To achieve the goal of determining the basis reflectance functions satisfied by these three constraints, we use a non-negative least squares matrix factorization solver. The result of

[3]This is explained in detail in section 3.2.4.

this analysis is a set of basis reflectance functions.

3.2.2.1 Matrix Factorization

The non-negative least squares (NLS) uses the large collection of texel reflectance function vectors as input, and finds a set of basis vectors that span the sub-space of the input vectors. In addition, the weights required to represent any of the input vectors will all be positive [62].

As with other analysis tools such as PCA, the computational requirements of NLS becomes prohibitive for a large number N of input vectors. Since we typically analyze the reflectance functions for a 1024×1024 grid of texels, we take a multi-step approach. First, we break up our texels into smaller blocks, where each block is typically 32×32 in size. Then, we run NLS for each of these smaller blocks, saving the top three selected basis reflectance functions from each block. This means that some basis reflectance functions might be lost during this stage. We then aggregate all of these selected reflectance functions and again perform NLS on this list. The most significant reflectance functions from this factorization become the set of basis reflectance functions S_B.

Other techniques exist for matrix factorization, such as principal components analysis (PCA). Consider, however, the following example. Imagine a texture that is a blend of red and green colors, and the spatial pattern of red versus green may be based on something like a band-limited noise function. Rather than recognizing red and green as the constituent components of the texture, PCA will return the vector yellow (the average of red and green) as being the dominant component. However, NLS will select red and green as the basis vectors. For further insights into various matrix factorization methods, see [60].[4]

3.2.2.2 Reflectance Function Sampling Details

The task of finding a set of basis reflectance functions relies on sampling the reflectance function. For each texel T_i and its corresponding sample point on the object (Figure 3.3), we sample its three dimensional reflectance function with a large number of input parameters

[4]We note that the matrix factorization method introduced in [60] (ACLS) might work here, too, however, some assumptions it uses are not necessarily true for procedural textures, and therefore might unnecessarily limit the number of procedural textures, so we use NLS.

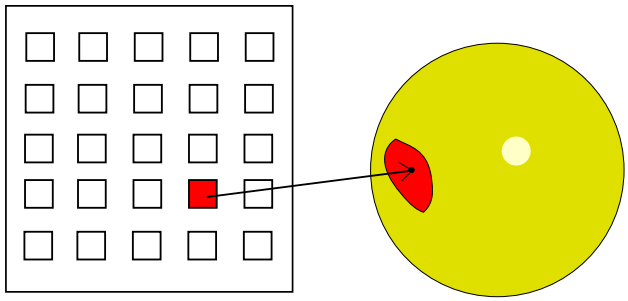


Figure 3.3: Texel/Surface Correspondence. Each texel in the procedural reduction map has a corresponding point and area on the surface of the textured object.

(incoming and outgoing directions). We will use the notation $R(\vec{\omega_i}, \vec{\omega_o})$ to describe a single reflectance function, with $\vec{\omega_i}$ and $\vec{\omega_o}$ describing the incoming and outgoing (reflected) light directions, respectively.

Each texel T_i corresponds to some reflectance function $R(T_i)$. Although each such reflectance function may be defined analytically, we sample a given $R(T_i)$ to create a sampled version of the function. We query the reflectance function using a variety of incoming and reflected directions $\vec{\omega_i}$ and $\vec{\omega_o}$ over the hemisphere given by the object's surface normal. We use the same set of incoming and outgoing directions for sampling each texel's reflectance function. Each color value that is returned is placed in the vector v_i that gives the sampled representation for the reflectance function. Thus, for n texels, there are n sampled reflectance functions v_i.

In principle, any number of samples of the reflectance function may be used. In practice, however, there is a balance between using few samples for speed and using many samples to accurately represent the reflectance function.

Table 3.1 lists the angles that we use to sample the reflectance function. To minimize the amount of sampling, we use the assumptions that the texture is isotropic and the specular component (if any) are located along or near the reflection vector. Isotropic reflectance functions can be described using three variables: incoming elevation (θ_i), outgoing elevation (θ_o), and difference in rotation around the pole between the two (ϕ). Note that the case where the outgoing direction is the normal, we have special sampling, but for cases where (θ_o) is non-zero, the sampling is the same and thus listed in the figure in the same line.

Table 3.1: Reflectance Function Sampling. Angles (in degrees) sampled for isotropic reflectance function matching and determination. First column is the incoming elevation, the second is the outgoing elevation and the last column is the rotational difference around the pole between the two. The last sample (bottom row) is used for transparency, and this gives us 35 total reflectance samples, each with a red, green, and blue component.

Incoming (θ_i)	Outgoing (θ_o)	Rotation (ϕ)
0	0	0
0	2	0
0	2	90
0	10	0
0	10	90
0	15	0
30, 45, 60, 85	θ_i	180
30, 45, 60, 85	θ_i	180 + 2
30, 45, 60, 85	θ_i	180 + 10
30, 45, 60, 85	θ_i + 2	180
30, 45, 60, 85	θ_i + 10	180
30, 45, 60, 85	15	180
30, 45, 60, 85	75	180
180	0	0

3.2.2.3 Transparency

Adding transparency requires only a little addition to the current sampling scheme. *Transparency* in this sense means filtered light with no refraction through the object (see Figure 3.4). One set of incoming and outgoing light directions are picked to determine if any light is filtered through the object, as shown in the last row of Table 3.1. If there is a transparent component to the texture, then this result will be non-black.

3.2.2.4 Automatic Detail Resolution

The sampling of a texel's reflectance function thus far has assumed that the reflectance function is constant over the corresponding area of the object. If this is not the case, then the super-sampling of that area should hopefully detect such differences. If detected, a texel will be further super-sampled accordingly until either the maximum number of super-samples is reached, or the average sampled reflectance function does not change with further sampling. Since the majority of such texels simply involve an edge, a maximum of sixteen super-samples is enough. Once the average reflectance function is found, it is used as that

(a) (b)

Figure 3.4: Transparency. (a) A procedural reduction map of a texture using transparency. Light rays are not refracted when entering the transparent object. (b) A procedural reduction map rendering of a bump mapped object.

texel's reflectance function.

However, if a procedural texture is complex enough, or if the super-sampling was poor, then the reconstruction of a good average of the reflectance function would also be poor. This problem is examined in detail in chapter 4.

3.2.3 Basis Weights

Now that the basis reflectance functions have been selected, an additional pass over the texels determines the basis weights. This step projects each texel's sampled reflectance function, v_i, onto each basis reflectance function vector to obtain the basis weights for the texel. The projection scalings are the corresponding weights for each basis reflectance functions. The projection of the vectors into that subspace also identifies vectors that are not in the subspace defined by S_B. We then add any of these reflectance functions that are outside of the space of reflectance functions defined by the current S_B by adding these reflectance functions to S_B. (These are the ones that may have been thrown out during the NLS processing step.) In practice, this does not occur often.

3.2.4 Basis Reflectance Function Representation

An integral part of the procedural reduction map is that each texel contains a weight for each S_B, the weighted sum representing the reflectance function for that portion of the textured object. The choice of representation for those basis reflectance functions, however, is independent of the procedural reduction map.

Several different approaches could be made for basis reflectance function representation. One example is spherical harmonics. This representation is satisfactory for certain types of reflectance functions, and has the advantage of being fairly fast to evaluate. Unfortunately, spherical harmonics have trouble capturing sharp highlights. Another approach might be to move to a refinable representation of spherical functions such as spherical wavelets [95]. Another possibility is to make use of a factored representation of the BRDF [61, 71]. Ultimately, however, no higher visual fidelity could exist than the original procedural texture itself.

Instead of creating a tabulated version of each basis reflectance function based on a large number of samples that might limit the visual fidelity to the original procedural texture, the procedural texture code itself should represent S_B. Given a basis reflectance function, B_i, as returned by NLS, we find the texel T_j whose sampled reflectance function is closest to B_i (normalized dot product of one). The algorithm records the corresponding position P_j on the object's surface as well as the surface normal N_j at this point. This position and normal, together with the procedural texture code, is the representation of the basis reflectance function B_i. During rendering, evaluation of B_i occurs by querying the procedural texture at this point.

Given that the representation of each B_i is the procedural texture itself with specific parameters, the reason for using NLS for finding the B_i becomes evident. In many instances, PCA and similar analysis methods often produce basis vectors that are nowhere near the original sample vectors. If such a basis vector is selected, there might be no texel that actually has the desired reflectance function. Non-negative least squares selects its basis vectors from the actual input vectors that are being analyzed. For our purposes, this means it will yield B_i that can be accurately represented by the reflectance functions of actual points on the surface of the object.

3.2.5 Surface Distribution of Normals

Both reflectance function analysis and rendering the texture from a procedural reduction map need surface normal information. The sampling of the reflectance function during

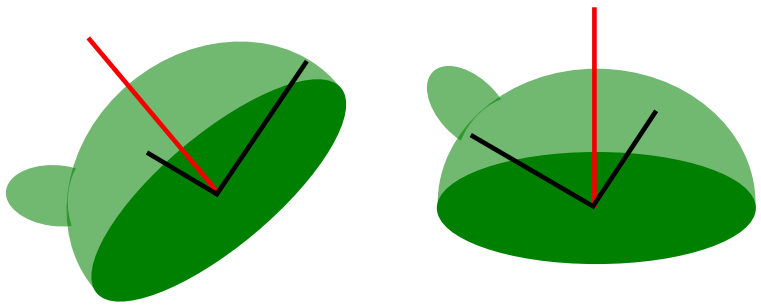


Figure 3.5: Recognition of Equivalent Reflectance Functions. The same reflectance function, but two different normals. Notice the sampling (black lines) are in the same direction, but get different answers.

analysis requires the surface normal so that the appropriate incoming and outgoing light directions can be used. For many procedural textures, the *geometric normal*, that is, the normal of the object at that point, is the surface normal used by the procedural texture.

In the case of procedural textures that perturb the normals during shading calculations, i.e., those that have bump maps or normal maps (Figure 3.4b), the system should automatically estimate the perturbed normals by sampling the reflectance function. The new normal after being perturbed by the procedural texture is often called the *shader normal*. However, for ease of discussion, we extend this term to include the geometric normal even when the procedural texture has no bump mapping.

When the shader normal differs from the geometric normal, sampling of the reflectance functions has to be done carefully. Transformation of the incoming and outgoing light directions need to be applied based on the normal that the texture will use in its calculation. Otherwise, using the geometric normal will result in equivalent reflectance functions not being recognized as such (see Figure 3.5). We need to automatically determine the shader normal so that we can fulfill the goal of minimizing the number of basis reflectance functions.

For textures that obey Helmholtz reciprocity ($R(\vec{\omega_i}, \vec{\omega_o}) = R(\vec{\omega_o}, \vec{\omega_i})$), Zickler et. al provide a way of calculating the true surface normal from six samples of the reflectance function [111]. Even though most real-world reflectance functions obey Helmholtz reciprocity, it is possible (and common) for textures to break this law. Therefore, this method does not help in the general case.

For the general case, we use a heuristic to determine the shader normal. Starting from

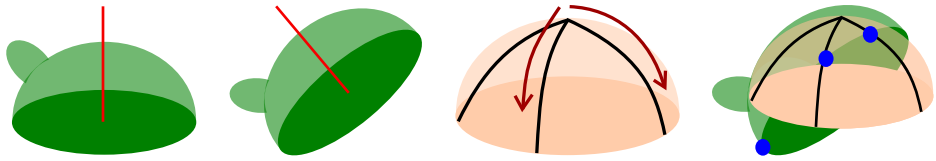


Figure 3.6: Sampling Shader Normal. (a) Reflectance function for geometric normal. (b) Reflectance function for shader normal. (c) Great arcs on the geometric normal hemisphere for sampling. (d) The points which are black. This is the plane defining the shader normal.

the geometric normal and heading toward the opposite point on the sphere, the algorithm adaptively samples the reflectance function along two distinct great arcs. The purpose is to find where the reflectance function is black, indicating a direction that is not in the outward-facing hemispheres of the surface.[5] As shown in Figure 3.6, the points of which the two arcs first reach the color black creates a plane (along with the center of the sphere) that defines the shader normal. This can be achieved with fifteen samples per line (thirty total) using a binary search (results are either black or not), giving an estimate of the shader normal that is within 10^{-4} radians of the original answer. This accuracy is sufficient for our method.

In order to sample the reflectance function along the great arcs, the incoming and outgoing light directions must be set. As shown in Figure 3.6, a sample point on the great arc specifies the direction for both the incoming and outgoing light directions.

At this point, the shader normal for a specific (u, v) coordinate has been determined. Since each texel has been super-sampled, there is a high probability that many texels will have multiple normals within the corresponding surface area. Therefore, such texels need a description of the surface curvature.

Following Olano and North's work with surface curvature, we use the *Normal Distribution Maps* [79] to describe the distribution of normals on the surface. In their scheme, they describe a distribution of normals on a surface as a Gaussian distribution. The description requires a mean normal vector (3D) and a 3×3 symmetric covariance matrix. The covariance matrix is the approximation of the distribution of surface normals on a sphere.

[5]This means that if a geometric normal is perturbed, the reflectance function cannot transmit light underneath the surface.

Such Gaussian distributions can be linearly combined by transforming each covariance matrix into the second central moment. Therefore, the super-sampled texel with various shader normals can be easily combined into one distribution, linearly combining the average surface normal and the second central moments. Furthermore, since second central moments can be linearly combined, weighted combinations of the distributions can also be created [109]. This fits our later needs, as will be shown in section 3.2.6.

Therefore, each texel's surface normal distribution is calculated by averaging the texel's super-sampled shader normals along with averaging the second central moments of those distributions. This creates a single, possibly broader, distribution that describes the texel's corresponding surface curvature.

Once the surface normal distribution for a texel has been determined, we store the shader normal distribution information at the given texel's location in the finest detail level of the pyramid. At this point, the finest level of texels are complete,[6] each one containing both basis weights and a Gaussian distribution of surface normals.

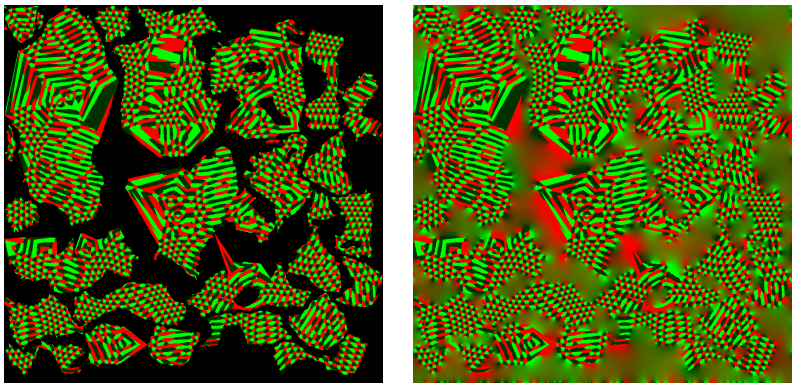
We will return to the surface normal distributions in section 3.3 when discussing rendering a procedural reduction map.

3.2.6 Filling the Higher Pyramid Levels

Now that the texels (with corresponding surface area) at the lowest level of the pyramid have been filled in, the remaining texels must also be filled. Just as with MIP-Maps, the higher levels of the pyramid will provide a good, fast approximation to the reflectance function for progressively larger areas of the surface. In this section, we will examine how to aggregate the information from filled texels to unfilled texels.

Similar to MIP-Maps and other Gaussian pyramids, each progressively higher level texel is determined by a weighted average of child texels. The procedure repeats until all levels of the procedural reduction map hierarchy are filled in. As noted previously, some portions of a texture atlas can be empty, that is, may not correspond to any portion of an object. See the black regions in Fig. 3.2(b) for an example of such regions. Sander et al. noted these

[6] This is technically not correct. Some texels do not have an associated surface area and are not filled in yet, but will be in section 3.2.6.



(a) (b)

Figure 3.7: Pull-Push Algorithm. (a) Before and (b) After the algorithm.

```
struct Texel {
    unsigned char weights[k];    // Basis weights
    float theta, phi;            // Direction of average normal
    unsigned char length;        // Length of normal (NDM)
    float covariances[6];        // Covariance matrix (NDM)
};
```

Figure 3.8: Texel structure. k is the number of basis reflectance functions. Elements listed as NDM (Normal Distribution Map) do not need to be included for some textures.

empty texels can be a problem if the (u, v) coordinates are near a patch boundary [92]. In addition, these empty texels should not contribute to texel values at higher pyramid levels. Their solution uses the pull-push pyramid algorithm of [38] to bleed the color information into the empty texels. We use the same pull-push technique, but where they applied it to color channels, we apply it to our basis weights and surface normal distributions. An example of before and after the pull-push algorithm can be seen in Figure 3.7.

We can easily apply a weighted linear combination of the basis weights for texel aggregation, and as noted earlier, so can the Gaussian distributions. Therefore, the pull-push algorithm applies in a straightforward manner to the procedural reduction map texel.

With all of the texels completely filled, the creation of the procedural reduction map ends by storing to a file the pyramid of texels along with the information to represent the set of basis reflectance functions S_B.

Table 3.2: PRM Memory Size. The second column lists the number (k) of basis reflectance functions in the procedural reduction map. The third column is the size of the PRM and the fourth column is the ratio of the PRM size to a MIP-Map of the same base size.

Texture	k	PRM Size (MB)	Ratio
Thresholded Feline	2	13.3	3.3
Shiny/Dull Scaled Dragon	2	13.3	3.3
Stone Bunny	3	14.7	3.7
Marble Igea	4	16.0	4.0
Bump Mapped Bunny	1	45.3	11.3
Transparent Bunny	1	12.0	3.0

3.2.7 Texel Data Structure

Figure 3.8 demonstrates the data structure for a texel in the procedural reduction map. Note that all data members marked as NDM (Normal Distribution Map) can be removed if the object is deemed smooth enough relative to the reflectance functions (which is true for functions that contain little or no specular components).

Note that the covariance matrix is stored as floats. Due to the close relationship between each of the matrix elements, further memory reduction is difficult while maintaining accurate reconstructions of the correct Gaussian distribution. However, further research might be able to compress this further.

Each texel has a memory cost of $k + 8$ bytes (where k is the number of basis reflectance functions in the procedural reduction map) when no normal distribution is needed. When the distribution is necessary, the memory cost becomes $k + 33$ bytes. The final numbers for the storage cost in this thesis are found in Table 3.2. All procedural reduction maps were created at the 1024×1024 size, and similar to MIP-Maps, use 4/3 of the storage required of the base size. Therefore, for models without covariance matrices, we averaged 14.7 MB per procedural reduction map (assuming three basis reflectance functions).

3.3 Rendering

As noted earlier, procedural reduction maps can be used with either a scan-conversion or a ray tracing renderer, and in this chapter, the implementation was for a ray tracing renderer. The only input to the procedural reduction map during the rendering stage is the (u, v)

coordinate and an estimate of the *texture footprint* (in texture coordinates).[7] Both of these inputs are computed by the renderer and are passed to the procedural reduction map.

The procedural reduction map takes the texture footprint and selects one of three possibilities. First, the texture footprint is small enough to use the original procedural texture with no anti-aliasing help from the procedural reduction map. Second, the texture footprint is large enough to completely render based on the procedural reduction map. Third, a linear combination of the first two should be used.

The texture footprint size is directly related to the size of a texel, giving an easy determination of which option to use. There are two important values when computing the correct option. First, the area of a texel at the bottom level of the pyramid. Second, the maximum number of super-samples for any given texel. For a given base level texel T_i, it has an area of a_i, computed as $\frac{1}{d^2}$, where d is the resolution of the pyramid's base level. The area corresponding to the minimal sampling is

$$a_m = \frac{a_i}{m_s} \tag{3.2}$$

where m_s is the maximum super-sampling of any given texel.

Therefore, if the area of the texture footprint, a_f, is less than a_m, use the original procedural texture. If a_f is greater than or equal to a_i, then use the procedural reduction map, otherwise use a linear interpolation of the two.

In the case that the procedural reduction map should supply an anti-aliased version of the procedural texture, the texture footprint area provides an estimate of how high in the pyramid to retrieve information from the texels. A linear interpolation from a_i to 1 (for the texel at the top of the pyramid) determines the height. Combined with the bilinear (u, v) interpolation inside a level, the algorithm computes the final location within the pyramid through trilinear interpolation.

The final result of the trilinear interpolation is a set of basis weights and a surface

[7] A complete discussion of the texture footprint and its implications will be discussed in chapter 4.

normal distribution, which can also be thought of as a new texel, T_f, describing the reflectance function and surface curvature corresponding to the texture footprint. There are two methods to compute final radiance, dependent on the surface curvature. We refer to these methods as the *mean normal approximation* and the *multiple samples approximation* methods.

3.3.1 Mean Normal Approximation

For fast rendering, we simply use the mean of the normal distribution as the surface normal, N, for shading. We evaluate the basis reflectance functions and scale each of them by the appropriate weight to achieve the final anti-aliased result.

Evaluation of a basis reflectance function depends on the representation of the basis reflectance functions. As described in section 3.2.4, each B_i is represented implicitly as a normal N_i and a position P_i on the object as input to the original procedural texture. Therefore, to evaluate a basis reflectance function, the procedural reduction map determines the rotation that transforms N to be coincident with N_i. The light and the view vectors are then rotated into the appropriate coordinate system. The procedural texture is then invoked at the location P_i with the transformed light and view vectors. The returned value from the shader is the outgoing radiance for that basis reflectance function.

Further discussion, including deficiencies, of the mean normal approximation method is discussed in chapter 4.

3.3.2 Multiple Samples Approximation

For cases where the normal approximation method fails, we resort to super-sampling in screen-space. Using the original rendering program, we super-sample this pixel. The number of super-samples can be determined in a variety of ways, but for our examples, we used 16 samples since this is what we are comparing against.

For a moment, consider surface normal distributions for flat surfaces. The covariances for the normal distributions will remain small even progressing through higher levels of the pyramid. Now, consider highly curved surfaces. Texels from even the lower levels must represent substantially curved portions of the surface, and thus surface normal distributions

become broad. The distributions also become broad in the case of bump mapping, since even the lower level texels get their normal distributions from many different normals that may cover a substantial portion of the Gauss sphere.

There are many cases that we can rely on the mean normal approximation instead of the multiple samples approximation. The choice is based on four attributes: the curvature of the surface, the intensity of the specular component, the shape of the specular lobe, and the current incoming and outgoing light directions.

Specifically, there are three cases when we can use the mean normal approximation: if the surface is nearly flat; the intensity of the specular component is small; or the shape of the specular lobe is glossy.[8] If any one of these cases is true for most of the procedural reduction map pyramid and its associated surface, then the normal distribution map is not even required and the mean normal approximation will be used every time.

However, when none of these cases apply, then the normal distribution map must be stored in the procedural reduction map and consulted each time the procedural reduction map is queried. In this case, the decision must be made whether to use the mean normal approximation or the multiple samples approximation. This decision is made per-pixel and follows the three cases mentioned previously. However, instead of testing all three cases, they can be generalized into one test. If the surface normal reflects most of the light toward the viewer, then use the multiple samples approximation. Although this will use the multiple samples approximation more than necessary, in practice it is sufficient.

To compute whether the surface normal distribution would reflect most of the light toward the viewer, consider the incoming and outgoing light directions, $\vec{\omega_i}$ and $\vec{\omega_o}$, respectively. In order for most of the light to reflect from $\vec{\omega_i}$ to $\vec{\omega_o}$, then the surface normal should be in the direction of the halfway angle:

$$H = \frac{\vec{\omega_i} + \vec{\omega_o}}{\| \vec{\omega_i} + \vec{\omega_o} \|} \tag{3.3}$$

[8] This topic is discussed in detail in chapter 4.

(a) (b) (c)

Figure 3.9: Multiple Samples Approximation: frequency of the multiple samples approximation. (a) 16 samples per pixel, no procedural reduction map. (b) Procedural reduction map, 1 sample per pixel (except for multiple samples regions). (c) Multiple samples regions are marked with red. During the animation, an average of 1.6 samples per pixel were used.

Therefore, to compute whether to use the multiple samples approximation simply depends on whether H lies within the surface normal distribution. Since the distribution is Gaussian, a cutoff must be used. For the examples in this chapter, that cutoff is 2.5 standard deviations, counting for a majority of the distribution. To calculate this, we invert the covariance matrix of the surface normal distribution for texel T_f and multiply by the H (after the center of the distribution has been subtracted out of it). If the result is farther than 2.5 standard deviations from the mean normal, then we use the normal approximation method instead.

The impact of using this method will be examined in detail in chapter 4. However, it should be noted here that of all the procedural textures used in this chapter, only the bump mapped bunny required this method. Furthermore, even in the case of the bump mapped bunny, few cases existed that required the multiple samples approximation (see Figure 3.9.)

3.3.3 Calculating the Texture Footprint

For the ray tracing renderer, we estimate the texture footprint area by shooting rays at the four corners of a pixel. When all four rays hit the same textured surface, the footprint area calculation is straightforward. Each ray/object intersection corresponds to a position in texture space, and we enclose the four given positions in a polygon that becomes our footprint.

In the event that two or more objects are hit, we shoot more rays that further subdivide

the pixel. The edges of the pixel are subdivided using a binary search until each edge has $\frac{1}{16}$ accuracy. From these additional rays, we estimate footprint sizes on the various objects, and determine what fraction of the pixel's color will be due to each object. This approach generates smoothly anti-aliased silhouettes of textured objects.

Textured mesh objects that have been unfolded into multiple patches in an atlas are also specially treated. If the four original rays for a pixel hit more than one texture patch in the same atlas, there is a danger of miscalculating the footprint size. For this reason, rays that strike different patches are treated just as though they hit separate objects, causing additional rays to be shot that then give the footprint areas and the contributing weights for the various patches.

CHAPTER IV

PROCEDURAL REDUCTION MAP ANALYSIS

This chapter gives an analysis of the procedural reduction map algorithm that is described in chapter 3. The sections that follow will describe different aspects of the algorithm. First, the central difficulties in anti-aliasing procedural textures will be discussed. Second, the assumptions made by the procedural reduction map about the procedural texture will be examined. Next, interesting details of the algorithm will be presented that were not important to the algorithm description in the previous chapter. Finally, results and conclusions are drawn at the end of the chapter.

4.1 Procedural Texture Anti-Aliasing Difficulties

Some of the difficulties in anti-aliasing procedural textures were mentioned in chapter 1, but are repeated here again for discussion.

Difficulty 1. Point versus Area Problem: *A procedural texture's input is an infinitesimal point which has no area, but anti-aliasing requires integration over an area.*

Since the procedural texture input has no mathematical area, integration over an area could theoretically require an infinite number of samples. Since an infinite number of samples is impossible, a completely automated method for anti-aliasing any procedural texture therefore could not sample, but would instead have to examine the code itself to for function analysis. However, with nothing known *a priori*, even then, examination of the code is outside of the realm of practicality.

Difficulty 2. 3D Problem: *Procedural textures are defined throughout three-dimensional space, making general anti-aliasing much more difficult.*

This problem is an extension of the first (Point versus Area Problem), but adds extra dimensionality to it. Specifically, the first problem concerns the problem of continuity,

whereas this the problem of dimensionality. Without solving the first, the problem of dimensionality cannot be solved, either. However, even if the first problem is solved, that does not guarantee a solution in higher dimensions.

Difficulty 3. Integration Problem: *Procedural textures can have complex reflectance functions that must be integrated corresponding to the incoming and outgoing light directions.*

Assume that an automatic method for anti-aliasing procedural textures exists that can overcome the Point versus Area Problem as well as the 3D Problem. However, during rendering, a reflectance function needs to be integrated over a region of the surface and over incoming light directions to achieve the correct result. The integration is usually at least three dimensions, which are independent of the three dimensions of the 3D Problem. Therefore, the current dimensionality (not including the Point versus Area Problem) of anti-aliasing general procedural textures problem is at least six dimensions! For a more complete discussion of this, see section 5.1.

Difficulty 4. Arbitrary Computation Problem: *Procedural textures can be defined as anything.*

Finally, the Arbitrary Computation Problem states that the dimensionality of the problem can be arbitrarily high, as well as arbitrarily obtuse.

4.1.1 Solving the Difficulties

However, despite the many historical difficulties in anti-aliasing procedural textures, procedural reduction maps solves, or at least ameliorates, the minification problem in most circumstances. The insight that many of the impossibilities and/or difficulties in automatic anti-aliasing procedural textures can usually be either ignored or circumvented directly led to the discovery of procedural reduction maps.

The name itself is derived from its source: *procedural* textures; its algorithm: complexity *reduction*; and the similarity in its final appearance to MIP-*Maps*. The method draws upon related work in three ares. Since the analysis for procedural reduction maps closely

resembles the analyses of these other works, they are briefly discussed here.[1]

The first, most obvious, is the MIP-Map method found in [106]. Both the MIP-Map and procedural reduction maps store intensities of reflectance functions. However, procedural reduction maps also store the reflectance functions themselves, as well as information about normal distributions.

The basis reflectance functions of a procedural reduction map resemble the Inverse Shade Tree method, both in acquisition and rendering [60]. However, as stated earlier in chapter 2, the Inverse Shade Tree method is not in itself capable of achieving the goal of minification anti-aliasing. The method does not attempt to perform anti-aliasing and is different in three main ways: its input is a set of real world measurements, not artificial textures such as procedural textures; minification artifacts are not addressed; and procedural textures often have a combination of multiple reflectance functions at every point, which violates one of the assumptions of Inverse Shade Trees.

The third method is the *normal distribution map* found in [35] and [79]. This method, while key to procedural reduction maps, in itself does not address the acquisition of reflectance functions from procedural textures, nor the anti-aliasing of varying reflectance functions across a surface.

Procedural reduction maps take these concepts and adapts them to solve the automatic procedural texture anti-aliasing problem. To do so, certain assumptions must be made to handle the difficulties listed earlier. The next section examines these assumptions.

4.2 Assumptions

Procedural reduction maps use carefully chosen assumptions that, while limiting the space of procedural textures that can be anti-aliased, still retain a large majority of the common procedural textures.

There is no delineation of what makes a procedural texture useful or not. Rather, each user has different needs and will have correspondingly different views on what is commonly useful and what is not. Despite this, during the ensuing discussion of the assumptions for

[1] The discussion is brief since their concepts have already been discussed in previous chapters.

procedural reduction maps, the reader should keep in mind this issue of *texture usefulness* with relation to procedural textures that can and cannot be anti-aliased with this method.

The following are assumptions for procedural reduction maps along with a discussion for the inclusion of that assumption, ranging from the practical to the absolutely necessary.[2]

Assumption 1. Light Properties: *The input texture obeys, or at least, generally obeys, the properties of light. That is, the resulting color is a function of the view direction and the incoming lights.*

Assumption 2. Distant Light: *During rendering, the light is located relatively far from the object.*

Assumption 3. Light Sampling: *The input texture cannot sample the scene beyond lights and shadowing.*

Assumption 4. Specular Lobe: *The input texture cannot have specular lobes outside of the reflection vector.*

Assumption 5. Isotropic Reflectance Functions: *Input textures cannot use reflectance functions more complex than isotropic BRDFs (plus non-refracting transparency).*

Assumption 6. Non-Deformable Model: *Input models must be non-deformable.*

Assumption 7. Time-Invariance: *The input texture must be time-invariant.*

Assumption 8. Displacement-Free: *The input texture must not use displacement mapping.*

4.2.1 Light Properties

The Light Properties Assumption does not mean that the procedural texture must obey the laws of physics with regards to optics. Rather, it means that the procedural texture should approximate the laws of physics, and that the closer the approximation is, the better the procedural reduction map algorithm will perform.

[2]These assumptions apply only to procedural reduction maps as described in chapter 3. Future chapters will give up a little automation for removing some of these assumptions.

This is an essential assumption. It limits the output of the procedural texture to in some way match the input. From this fact, we can now handle the Arbitrary Computation Problem. Furthermore, it limits procedural textures to approximate the rendering equation (Equation 1.1) and the reflectance equation (Equation 1.2), thereby severely limiting the number of dimensions.

The interesting (and obvious) observation is that, while necessary from a decidability viewpoint, this assumption is also required from the *photo-realistic* viewpoint. Anti-aliasing a photo-realistic scene requires approximation of the rendering equation which in turn is a description of how light interacts with a scene. If a procedural texture violates this too much, the scene will not look "real." Therefore, this assumption not only provides decidability, it also matches the kinds of textures we desire to anti-alias. From this, it should be clear that this assumption omits very few useful textures.

4.2.2 Distant Light

Before examining this assumption, first consider the following. Given a planar region, if the light source and/or the viewer is close the surface, the incoming and outgoing light directions change rapidly over the area, despite the same normal at every point. However, in the case of minification, the viewer is distant, and the outgoing light direction is relatively constant over the region. If the light source is allowed to be relatively close to the object, then even planar regions can require sampling from vastly different portions of the reflectance function. Integration then must be done over the distribution of surface normals, the distribution of the incoming light, *and* the reflectance function itself. The cost of the added dimensionality for close lighting is too high.

Therefore, the light source is assumed to be distant relative to the object's surface. The Distant Light Assumption, though, is not actually that different from MIP-Maps, which assume the light is distant as well. Note that if this assumption is violated, procedural reduction maps still work, but have somewhat degraded visual fidelity.

4.2.3 Light Sampling and Specular Lobe

We require the Light Sampling Assumption and the Specular Lobe Assumption not for dimensionality reduction, but rather for *reflectance function detection* during the creation stage. Automatic detection of different reflectance functions can be arbitrarily difficult.

Consider two reflectance functions R_1 and R_2. If, for all (θ, ϕ), $R_1 = R_2$, then any sampling of R_1 and R_2 would match, yielding the correct answer that they are identical. However, if R_2 was different than R_1 at only one infinitesimal angle (θ_0, ϕ_0), it would be impossible to find the difference using only a finite number of samples. Similarly, two almost identical reflectance functions could sample the scene for global illumination in the same way except for one specific, infinitesimal angle, again yielding the impossible task of automatic detection of such differences. Therefore we limit these so that we may identify reflectance functions.

Note that these assumptions, since they are related to difference detection, do not mean that the procedural reduction map algorithm will not work for procedural textures that violate these assumptions. However, the visual fidelity of the anti-aliased result could be arbitrarily lower than textures that don't violate these assumptions.

4.2.4 Isotropic Reflectance Functions

Unfortunately, the problem of reflectance function detection also affects the ability to determine higher dimension reflectance functions. From this, the Isotropic Reflectance Function Assumption eliminates reflectance functions such as anisotropic and BSSRDFs (among others).

Anisotropic BRDFs can have arbitrary differences in the reflectance function and therefore can be statistically impossible to recognize as being anisotropic. Even when identified as anisotropic, recognition that two reflectance functions are equivalent requires an infinite amount of sampling.

BSSRDFs, along with refracting transparency, involve extremely high dimensionality, including not only incoming and outgoing light directions, but also light interactions with the geometry. Although access to the object exists, each point on the object would require

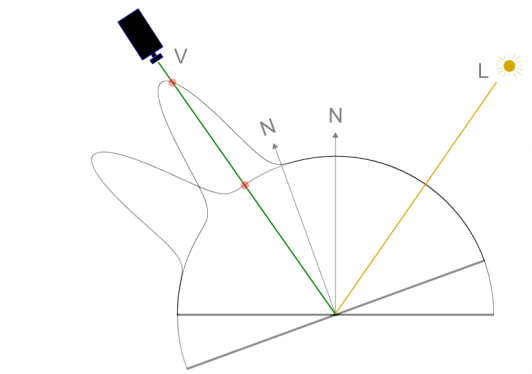


Figure 4.1: Reflectance Function Instantiation. The same reflectance function is shown, but with two different normals. The incoming light direction, L, and the outgoing light direction, V, are constant. This can occur when the surface is rough. Notice the vast difference between the reflectance function values (shown by the red dots.)

analysis and storage of at least four dimensions. Integration over an area would be even more expensive than the anisotropic case.

This is the most limiting of all the assumptions. Many useful procedural textures fall outside of this assumption. Fortunately, they are still much less common than isotropic textures.

4.2.5 Non-deformable

Deformable models can arbitrarily change the distribution of surface normals, thereby arbitrarily changing *reflectance function instantiation*. Consider two points, P_1 and P_2, with the same reflectance function R, but with two different normals N_1 and N_2. If P_1 and P_2 are close enough together, and the light source and light destination are far enough away, then the incoming light direction, L, and the outgoing light direction, V, are the same for both points. The result of applying V and L to R at P_1 gives a different result compared to applying them at P_2 because they have different normals. We call P_1 and P_2 two different instantiations of the reflectance function R. This can be seen in Figure 4.1.

Because deformable models can arbitrarily change the surface normal distribution, and

thereby arbitrarily change the reflectance function instantiation, we add this assumption to keep the surface constant.

4.2.6 Time Invariance

Even though the dimensionality of the procedural texture has been drastically reduced, time-varying textures provide a dimensionality that can easily be prohibitive in anti-aliasing automatically. Consider a procedural texture at some point P. P can have an arbitrarily high number of reflectance functions in a single unit of time. The next unit of time can have a completely different set of reflectance functions, again with an arbitrarily high number. Therefore, this assumption removes this problem.

Note that this assumption does remove some useful procedural textures. However, since most objects do not exhibit time-varying appearances, most of these are uncommon. Furthermore, the goal of this thesis is the anti-aliasing of minification artifacts, not temporal artifacts.

4.2.7 Displacement-Free

Although procedural reduction maps handle bump mapping, it does not handle displacement mapping. Procedural textures that use bump maps are not actually affecting the geometry of the object, whereas displacement mapping does just that. Displacement mapping, however, adds the noticeable effect of changing the geometric silhouette of the object. The aliasing that occurs from this aspect is not texture minification aliasing but rather falls under the category of silhouette aliasing. The shifted normals from displacement can be handled, but not the geometric displacement. Note that the violation of this assumption would only leave the geometric silhouette aliasing problem; the minification anti-aliasing of the displaced normals would still be handled.

4.3 Procedural Reduction Map Creation

In this section, the interesting details from the creation of the procedural reduction map will be discussed. Therefore, not all sections from the previous chapter will be discussed, only those that have non-trivial aspects.

4.3.1 Texel/Sample Point Correspondence

As stated in the Point versus Area Problem, one of the largest difficulties in anti-aliasing procedural textures is integrating over an area. The input of a procedural texture is an infinitesimal point, which has no area. Despite this, high quality anti-aliasing must make a good approximation to the procedural texture's response to the texture footprint.

Using point sampling, the problem of an infinitesimal point is addressed by conversion to an area for each texel. The conversion from a point input to an area input is critical to all portions of the procedural reduction map: the normal distribution map represents an area of the object's surface; each texel, which contains reflectance function weights, also corresponds to an area of the surface.

Furthermore, we also address the 3D Problem through point sampling only along the object's surface. Even though the surface is in 3D, it can be "unwrapped" onto a plane. By applying the procedural texture at the object's surface, we limit the dimensionality of the anti-aliasing problem, even if the procedural texture is defined in three dimensions.

The difficulty in point sampling the procedural texture according to the texels and the associated points on the surface lies with fact that the bottom level of the texel is finite, while the procedural texture itself can have infinite detail with infinite frequency. Quite conceivably, a procedural texture might suffer from minification aliasing at all resolutions. However, this does not impact the effectiveness of procedural reduction maps compared to current methods since screen-space super-sampling would fail as well. In fact, while the texture footprint is large enough, the procedural reduction map would generate little or no minification aliasing artifacts. Visual fidelity would suffer according to the degree difference between the sampled texels and the actual representation, which due to the arbitrarily high complexity cannot be bounded on its error. However, as stated previously, super-sampling would also suffer, but would alias in addition.

Despite this, the practical reality of a large visual error due to infinite detail with a large frequency in a procedural texture is small. The following must be true for there to be a problem:

- The procedural texture must have infinite detail.
- The detail must have a frequency greater than half the sampling rate of the area covered by the point sampling of the surface.
- The reflectance functions represented in a texel are grossly different in the specular lobes.

From these facts, this issue is a minor concern and is not even listed as an assumption. It is not referred to again in the rest of this thesis.

4.3.2 Basis Reflectance Functions

There are several issues concerning the basis reflectance functions. Among them are the number of basis functions and the identification of basis functions.

4.3.2.1 Number of Basis Reflectance Functions

One of the central ideas to the procedural reduction map is that there only a few reflectance functions used throughout a procedural texture. Here, we will examine such a claim.

Consider a typical procedural texture with varying diffuse and specular components. For each component, there are a maximum of three basis reflectance functions describing them since there are three different color channels. Most specular components are even simpler since many use white as the base color for the specular lobe. The resulting number of basis reflectance functions is governed then by the following equation:

$$k = 3d + s \tag{4.1}$$

where k is the number of basis reflectance functions, d is the number of different types of diffuse reflectance functions (i.e., excluding scaling) present in the procedural texture, and s the number of different specular lobes present.

Now consider that a given procedural texture has many reflectance functions, i.e., the texture has a blend of specular lobes. In truth, the human eye would have trouble distinguishing between small differences in the specular lobe, especially on a curved object. Even

Figure 4.2: Sampling: Isotropic vs. Anisotropic Coordinate Systems. The normal, N, the outgoing light direction, V, and the incoming light direction, L, are all known. (a) Isotropic BRDF. The three parameters $(\theta_i, \theta_o, (\phi_i - \phi_o))$ can be set since they depend only on N, L, and V. (b) Anistropic BRDF. The four parameters $(\theta_i, \phi_i, \theta_o, \phi_o)$ can only be partially set. θ_i and θ_o are known since N, L, and V are known, but ϕ_i and ϕ_o cannot be set since the anisotropic axis (blue arc) is unknown.

if visual fidelity to that kind of texture were lower, the noticeable differences by the human visual system would also be slight.

Given these items, and the wide range of procedural textures used later in the chapter, we assert that this is a reasonable assumption.

4.3.2.2 Identification of Basis Reflectance Functions

There are two interesting aspects to identifying the basis reflectance functions. First, is the low number of samples to identify unique as well as equivalent reflectance functions. The second one is how the parameters are passed to the procedural texture during reflectance function sampling.

The number of samples with regard to reflectance function sampling is small (35). The principle at work is the minimization of the number of samples while still capturing the key aspects of the reflectance function. We can use a small number of samples directly because of the assumptions. Specifically, the Specular Lobe Assumption allowed minimal samples to capture the specular lobe since we knew where it was located. The Isotropic Reflectance Function allowed sampling in three dimensions as opposed to four, and with a known coordinate system (see Figure 4.2). Furthermore, from these assumptions plus the Light Sampling Assumption and the Light Properties Assumption, we knew that if two reflectance function sampling vectors were equivalent, so were their reflectance functions.

The second interesting item is how parameters are passed to the procedural texture. For example, the procedural texture could very easily attempt to take into account shadowing

Figure 4.3: Occlusion on Reflectance Function Identification. (a) Part of the object occludes light coming from certain directions. (b) The resulting sampled reflectance function (black) differs from the actual reflection function according to the occlusion (differences shown in red).

and thus would sample the scene for occlusions. However, if any occlusions are allowed, then two identical reflectance functions could easily appear to be very different in their sampling (Figure 4.3). Therefore, all occlusions must be disallowed from outside of the procedural texture (so that we can keep treating the procedural texture as a black box.) Even if there are no other objects in the scene, self-occlusion can happen.

To handle the problem, the renderer always returns false for the shadow test. Furthermore, the renderer maintains the incoming and outgoing light directions as requested by the procedural reduction map creation algorithm.

4.3.2.3 Matrix Factorization

The goal during matrix factorization is the identification of the basis reflectance functions. However, as stated in section 3.2.2, there are several sub-goals that will generally help the algorithm. One of these sub-goals was that each final basis reflectance function must be found among the sampled reflectance functions. Another sub-goal was to minimize the number of basis reflectance functions and the third was that the vectors should be as orthogonal as possible for reasons of numerical accuracy.

The first sub-goal, distinct basis reflectance functions, is due to the representation chosen. Since the basis reflectance function representation is the original procedural texture itself, then the result of the factorization has to be a pre-existing vector.

The second sub-goal, minimizing the number of basis vectors, is directly related to compression. The number of bytes needed for each texel grows linearly with the number of basis weights that are stored.

The last sub-goal deals with the projection of the sample vectors onto the basis vectors. During the basis weight computation, new basis reflectance functions might be found. However, there had to be a cutoff between actual new functions and simple numerical inaccuracies. This factor is ameliorated by selecting original basis reflectance functions that are as orthogonal as possible.

4.3.3 Surface Distribution of Normals

During shader normal determination, the reflectance function for a texel is repeatedly tested, looking for the orthogonal plane to the shader normal. In practice, some reflectance functions go to black before this plane. (We call the plane determined by the origin and the two great arcs' black points a *false plane*.) This can be tested by choosing a third arc on the sphere and testing near the plane. If it fails to have color, then a third search going up the arc toward the geometric normal will eventually find another point where color no longer appears. The three points on the three great arcs now determine the orthogonal plane for the shader normal.

If the Displacement-Free Assumption is violated and displacement mapping used, then this method for intercepting false planes still works. Since the reflectance function samples are far away, the displacement of the point is inconsequential. Furthermore, the perturbed normal is treated just as it is in the normal bump mapping case.

4.4 Procedural Reduction Map Rendering

Like the section that discusses the creation of procedural reduction maps, this section will examine some of the interesting details of rendering with procedural reduction maps.

The first such detail involves rendering if displacement mapping is used. If the Displacement-Free Assumption is violated, then during rendering the silhouette of the object will not be the same as the true textured object. However, for many levels of the procedural reduction map, there will be no noticeable difference. The scale of displacement mapping is small, and the procedural reduction map works only when the object itself is small, when small scale geometric differences are indistinguishable.

Continuing with the discussion of a violation of the Displace-Free Assumption, the basis

reflectance functions probably will use displacement mapping as well. However, this does not affect any rendering since the procedural texture is not given a light source, but rather a light direction. Similarly, the viewing direction is given. Therefore, perturbations to the point do not affect the rendering of the reflectance function.

Another item of rendering is shadows. Although shadows play a large and important role in photo-realistic rendering, their importance is, for the most part, orthogonal to the discussion and validation of this algorithm. Simple shadows are used in the accompanying videos and in the figures and therefore might display some aliasing. However, since this is a geometric problem and because there are few shadows in the scenes, they were not discussed or anti-aliased.

Object silhouette aliasing, however, was hard to ignore, and therefore the silhouettes of objects with procedural reduction maps were super-sampled. This should not be considered "cheating," since the silhouette aliasing artifacts were not part of the goal in anti-aliasing, as well as all running times for the procedural reduction map were slowed because of this.

4.4.1 Mean Normal Approximation

This approximation to reflectance function integration will obviously have shortcomings in many areas. However, before that discussion begins, it should be noted that MIP-Maps follow this exact approximation.

Using the average normal, combined with the Gaussian blending of texels, does a sufficient job of approximating the true integration of the reflectance function for many low dimension cases. This approximation, though, does not capture well certain aspects of illumination.

The first aspect is a rough surface. The specular highlight will actually blur out much more quickly when the texture is minified. For example, consider two texture planes, each with the same reflectance function. One has bump mapping and has a very rough surface, while the other has no bump mapping and is thus very smooth. If the incoming and outgoing light directions are place directly above the surface in each of those cases, then the brightness on the rough surface should be less than the smooth one since there is less

surface to reflect the light. This aspect applies to all highly curved surfaces.

The other aspect is that the values of the reflectance function over a distribution can be widely varying due to intense specular highlights. Such differences, even if they are not enough to cause aliasing, do degrade visual fidelity to the correct reflectance function integration. However, as noted previously, for many cases this is acceptable as it is with MIP-Maps. The next chapter will discuss NBRDFs, a solution to the reflectance function integration problem.

4.4.2 Multiple Samples Approximation

Recall that during rendering, if the surface is rough enough, and the reflectance function is highly specular, then instead of using the procedural reduction map, screen space multiple samples were used to approximate the reflectance function integration.

First of all, note that when this case arises, it is exactly the same as the original automatic anti-aliasing algorithm of super-sampling. With the exception of a very small overhead involved in the procedural reduction map algorithm, the costs are the same.

Second, this case does not occur often. Among all procedural textures used in the chapters up to this point and the first accompanying video, only the highly specular, highly curved, bump mapped bunny required the multiple samples approximation. Furthermore, even this case required the multiple samples approximation only a small fraction of the time (see Fig. 3.9). In the animation sequence of the bump-mapped bunny, an average of only 1.6 samples per pixel were required. This number is small because the specular highlights only occur in certain areas, and those can be determined by comparing the reflection vector to the normal distribution. [3]

The final observation of this method is that, despite intuition, our animations alias less than or equal to super-sampling at every pixel. Obviously, in pixel regions where the screen-space approximation method is used, then there is an equal amount of aliasing. Therefore, the interesting comparison occurs in the regions where the normal approximation method is used versus super-sampling. The normal approximation does not involve the specular

[3]This is valid due to our assumption that the specular highlights are only found along the reflection vector.

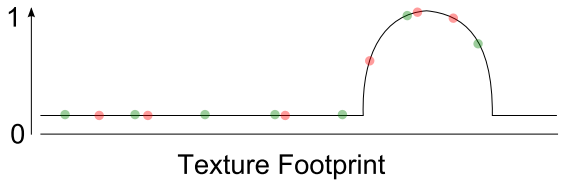


Figure 4.4: Super-Sampling Problem. The x-axis is the texture footprint. The y-axis is the color intensity, from 0 (black) to 1 (white). The red dots and the green dots represent two different samplings. Notice that the green will do a good job of determining the average color, but the red dots will get a significantly higher resulting color. This will result in aliasing.

component, even though it is possible there is a small portion of the normal distribution that does include it. From frame to frame, the color changes gradually due to the nature of the pyramid of averaged normals. On the other hand, super-sampling does not have access to such information. When a specular region takes up only a small fraction of a pixel, super-sampling may by chance occur for a disproportionately large number of the samples. Since the specular component typically has a much higher intensity, this sudden increase in color intensity that will disappear next frame is seen as aliasing. An example of this is in Figure 4.4.

4.4.3 Calculating the Texture Footprint

Calculating the texture footprint is not easy. Many renderers spend a lot of time approximating the texture footprint, and many get it grossly wrong even after much computation [33]. However, to validate the procedural reduction map algorithm, much care was taken to get a good approximation of the texture footprint. This is why pixels were subdivided and re-sampled when along patch boundaries. Obviously, poor approximations to the texture footprint will severely degrade visual fidelity, just as with MIP-Maps.

4.5 Results

The approach tested on a single-processor Intel Pentium 4, 2.8 GHz machine with 2 GB of memory. The renderer was a modified version of POV-Ray. The choice to use a public-domain rendering engine was so that during rendering we could easily catch and modify

procedural texture calls to use procedural reduction maps. There should not be any conceptual barrier to using procedural reduction maps in other renderers as well. For all examples, no hand anti-aliasing occurred, either in the code or after image generation.

Figure 4.5 demonstrates several procedural textures of varying types that are common in computer graphics: thresholded noise, a regular (hexagonal) pattern with differing specular components, cellular texture and marble. The figure demonstrates the textures at varying levels of zoom (each perfectly anti-aliased.) The figure also demonstrates what kind of aliasing can occur when nothing is done (1 sample per pixel), super-sampling is performed (16 samples per pixel), and when the procedural reduction map is used with only 1 sample per pixel. For the aliasing demonstrations, the object is far away and then the resulting image is magnified.

From left to right in Figure 4.5, each texture demonstrates a common and difficult aspect of automatic anti-aliasing of procedural textures. First, the thresholded noise texture applied to the feline model (10K triangles). This is difficult to anti-alias due to the discontinuous function at the boundary between blue and yellow. Although wavelet noise [27] has greatly improved the anti-aliasing of noise, it does not help in this situation.

The second texture is the hexagonal pattern applied to the dragon model (20K triangles). This pattern has three different reflectance functions, but what makes it most interesting is that one of these has a much different specular component. Traditional MIP-Maps cannot handle this type of texture, and this type of texture is often left as-is and super-sampled for anti-aliasing purposes.

The third texture is a cellular (Worley) texture applied to the bunny model (10K triangles). Cellular textures are popular, but they can be difficult to anti-alias. Anti-aliasing this texture by hand would be difficult at best and trying to make this a traditional MIP-Map loses the magnification benefits of the original texture.

The fourth texture is another common form of procedural texture, the solid texture marble, applied to the igea head (10K triangles). The white regions of this texture have been made more glossy than the dark portions. The difficulty with this type of texture is the non-linear correspondence between the noise function and the color space [33], as well

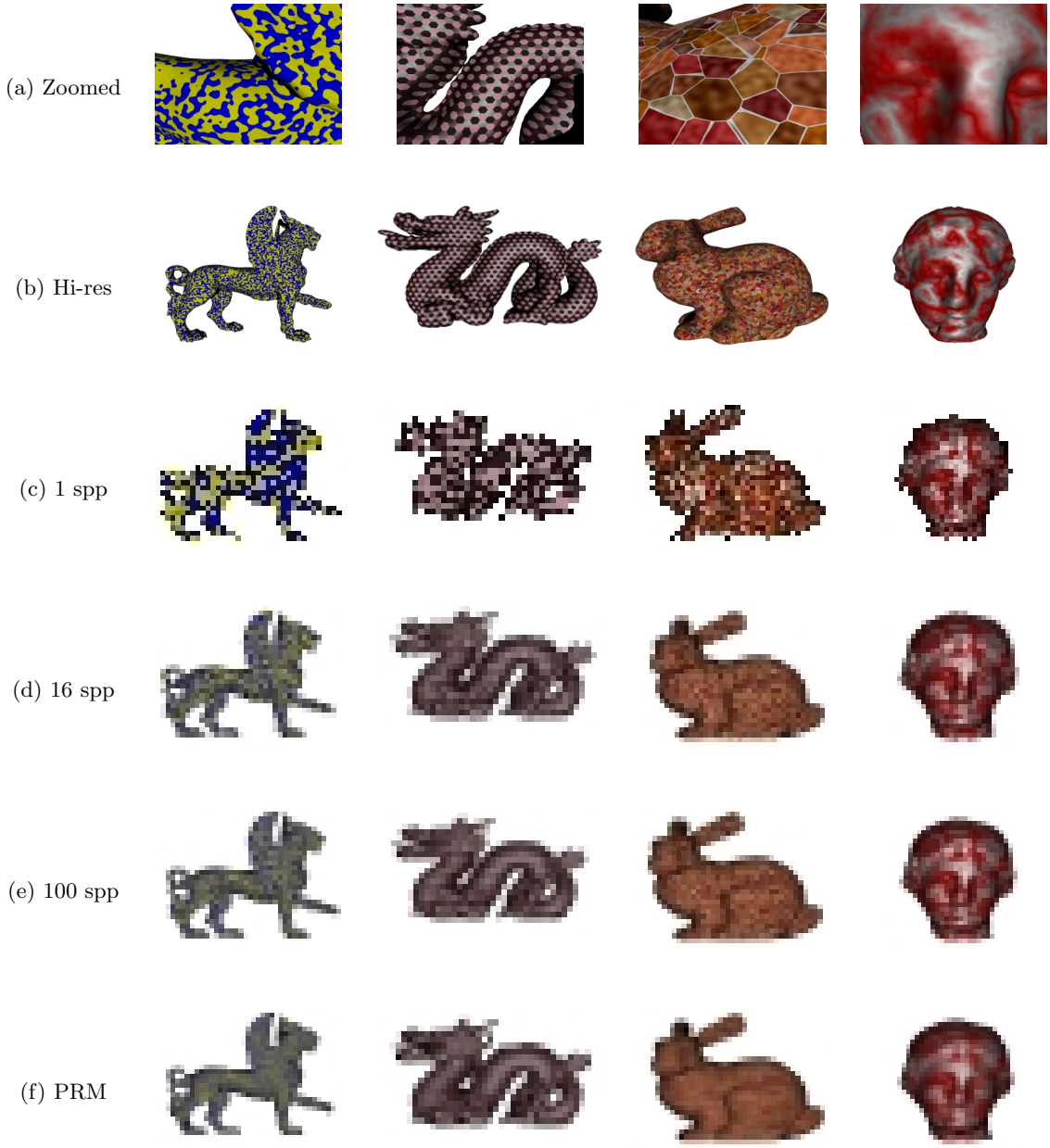


Figure 4.5: Procedural Texture Anti-Aliasing Results. Four representative textures are shown. The first two rows demonstrate the texture with very high quality renderings. (a) demonstrates the texture when magnified. (b) shows the texture on the entire object when the object is close enough such that there is no aliasing. For the next four rows, the object is placed far from the viewing screen and then rendered. The resulting image is then enlarged without interpolation. (c) one sample per pixel rendering, (d) sixteen samples per pixel, (e) is close to the ideal image (100 samples per pixel), and (f) is rendered using the procedural reduction map with only one sample per pixel.

Table 4.1: Numeric Difference Comparison. This table is the numerical counterpart to Figure 4.6 and Figure 4.7. The first column is the texture and the number of samples procedural reduction map (PRM). The second column is the RMS Error when compared against a highly super-sampled version (100 samples per pixel.) The third column is the Sarnoff average error and the last column the maximum Sarnoff error.

Texture	RMS Error	Sarnoff Average Error	Sarnoff Maximum Error
Feline (1 spp)	0.01497840	0.564944	4.336612
Feline (16 spp)	0.00052506	0.073782	0.875421
Feline (PRM)	0.00051552	0.117776	0.762443
Dragon (1 spp)	0.00821330	0.374346	2.902288
Dragon (16 spp)	0.00020464	0.051091	0.491755
Dragon (PRM)	0.00075209	0.112820	1.165156
Bunny (1 spp)	0.00772024	0.432233	3.001496
Bunny (16 spp)	0.00019565	0.060802	0.713868
Bunny (PRM)	0.00021114	0.074881	0.915190
Igea (1 spp)	0.00334340	0.235597	2.220521
Igea (16 spp)	0.00005903	0.028558	0.256852
Igea (PRM)	0.00038591	0.122338	0.825404

as the blending of a glossy component into portions of the texture.

Finally, we note that these textures were picked to demonstrate common procedural textures. Therefore, no attempt was made to make them "extra complex." Many procedural textures will be more complex than the ones listed here; varying reflectance functions and deeply nested code will make anti-aliasing them even more troublesome than these examples if done by hand.

Figure 4.6 and Figure 4.7 demonstrate the visual fidelity and anti-aliasing results in a more quantifiable manner. Through an implementation of the Sarnoff JND (Just Noticeable Difference) Visual Model and the root mean square error (RMS Error), both a visual and a numeric estimate of the difference between the actual answer, one sample per pixel, sixteen samples per pixel, and the procedural reduction map can be ascertained. Similar to MIP-Maps, blurring occurs of the original image, as can be seen in the Sarnoff difference images. However, these differences tend to be smooth, resulting in less aliasing during animation sequences. Table 4.1 lists the numeric equivalent of the visual differences.

We refer the reader to the accompanying video clips to see the quality of these results for animated sequences. The differences are even more pronounced in animated scenes. In

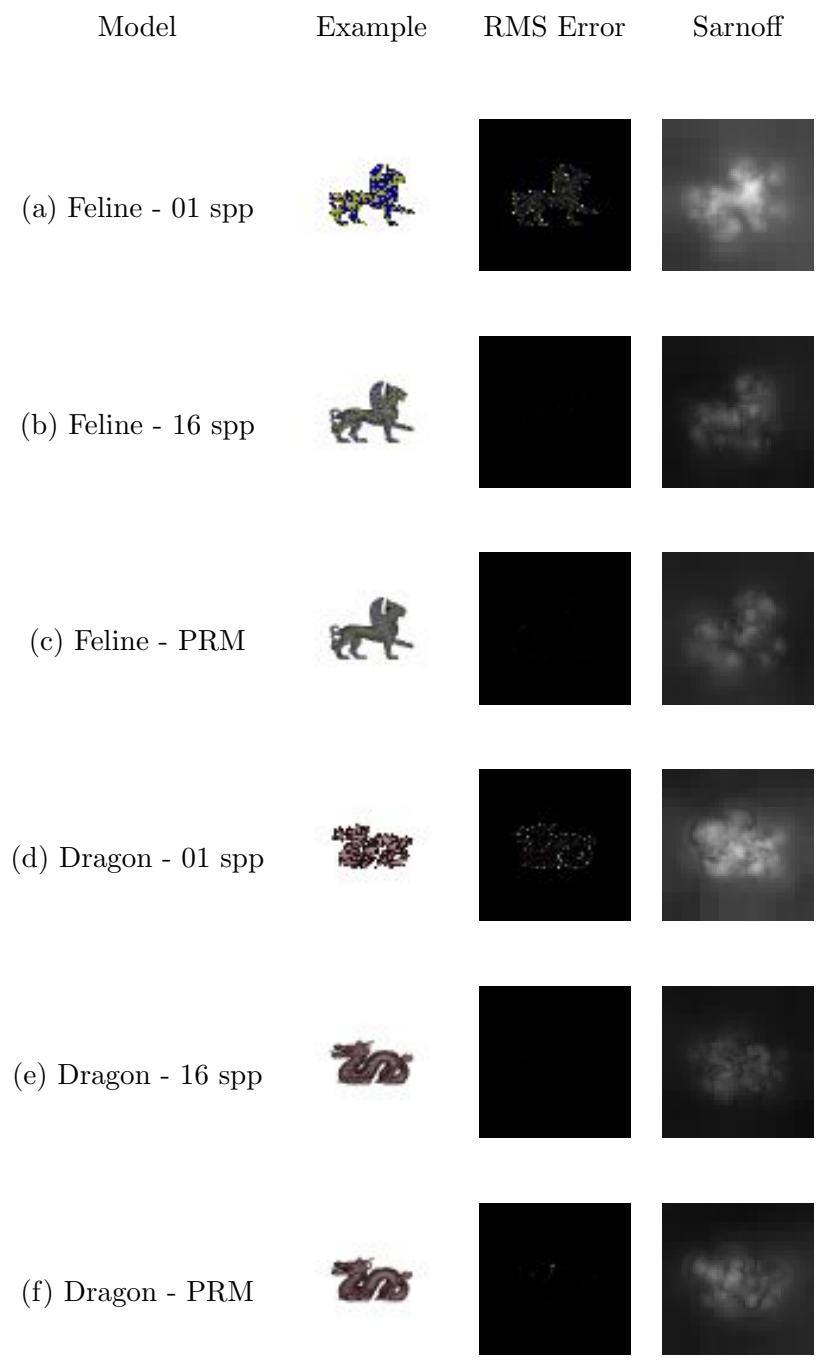


Figure 4.6: Visual Difference Comparison of Feline and Dragon. Using the RMS Error and the Sarnoff JND Visual Model, the differences between a near-perfect answer and each of the trials from Figure 4.5 can be visually compared. The first column is the name of the texture. The second column is the original picture used for comparison against the near-perfect. The third column is the RMS Error difference, and the last is the Sarnoff error difference.

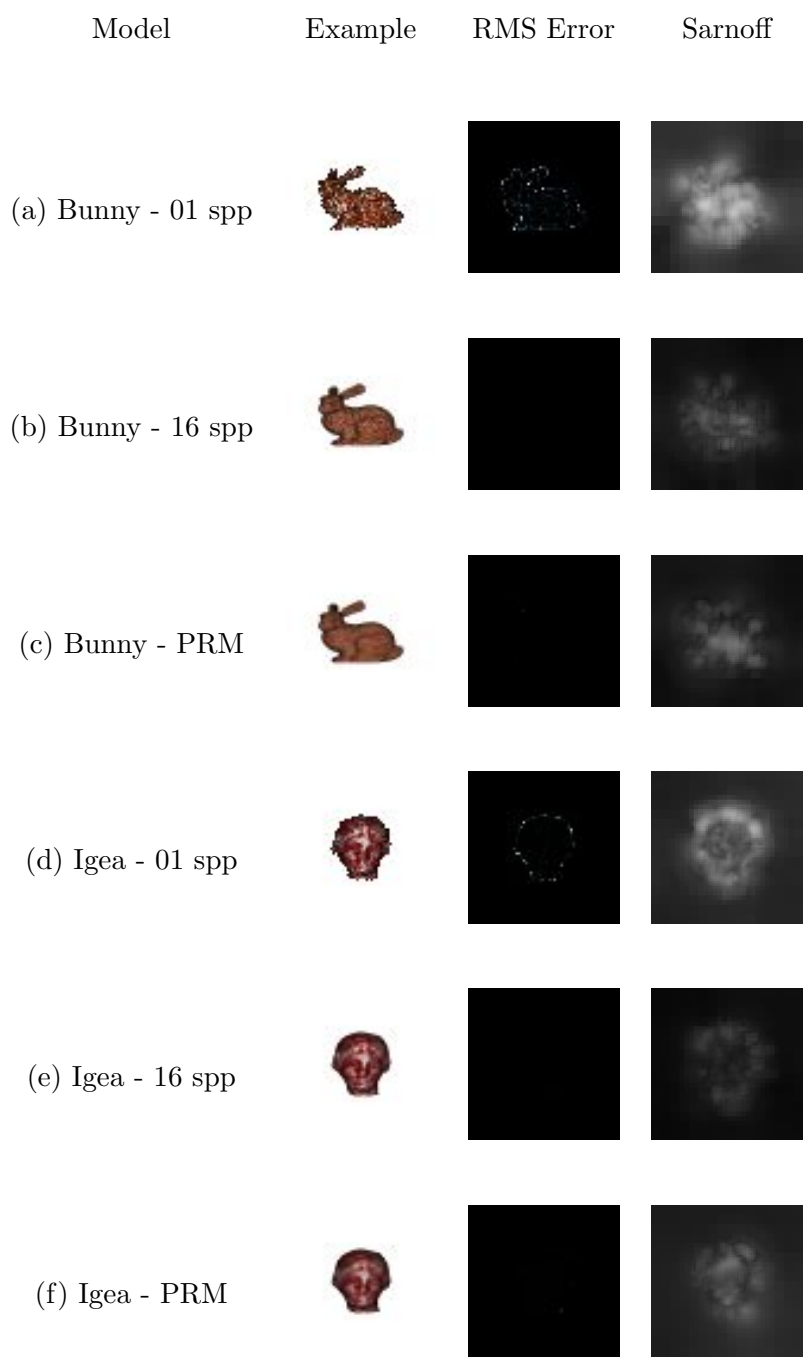


Figure 4.7: Visual Comparison of Bunny and Igea. Using the RMS Error and the Sarnoff JND Visual Model, the differences between a near-perfect answer and each of the trials from Figure 4.5 can be visually compared. The first column is the name of the texture. The second column is the original picture used for comparison against the near-perfect. The third column is the RMS Error difference, and the last is the Sarnoff error difference.

Table 4.2: Procedural Reduction Map Creation Times. Timings (hours:minutes:seconds) for creating procedural reduction maps of various textures. S_N refers to the time it took to determine the shader normal. "Sampling" is the time required to sample the reflectance function for each base-level texel. "NLS" is the time it took to run the non-negative least squares. The time it took to fill in each texel's weights and normal distributions, are found in "Filling". The total time to run the algorithm is in "Total". Note that with the exception of the bump-mapped texture, all textures had a function to return the shader normal.

Texture	S_N	Sampling	NLS	Filling	Total
Feline	0:00:00.1	0:01:16.1	0:13:45.2	0:02:09.3	0:17:52
Dragon	0:00:00.0	0:01:17.7	7:37:53.4	0:02:05.6	7:41:58
Bunny	0:00:00.1	0:02:36.1	7:03:19.4	0:02:20.0	7:08:58
Igea	0:00:00.1	0:02:20.9	6:31:23.6	0:03:00.4	6:37:06
Bump Map	0:01:45.4	0:01:48.3	0:03:44.6	0:02:17.0	0:09:52
Transparency	0:00:00.1	0:00:43.5	0:03:41.2	0:02:20.1	0:07:02

particular, the 16 samples per pixel videos alias noticeably. We also note here that this algorithm does not handle silhouette aliasing, but rather minification aliasing.

Table 4.2 gives timing results for the procedural reduction map creation, along with timings for various parts of the creation portion of the algorithm. The NLS procedure is by far the most expensive aspect of the algorithm, but we note that for many textures, such as the cellular texture, that finding the set of basis reflectance functions B is difficult even for humans. We also note that the purpose of this algorithm is for improving complex procedural textures used in animations. Texture creation and texture use in an animation are often separated by days or longer. In this typical case, the texture author might have spent many hours or even days writing the texture. At the end of the day, the author could run the creation algorithm and it would be finished well before she came back the next day.

4.5.1 Timings

Table 4.3 gives the rendering timing results for several textures, some with a corresponding procedural reduction map and some without. The two main costs in rendering are the number of rays cast and the number of texture calls. The table attempts to delineate the costs and benefits of our algorithm. In the animation, notice that visually the procedural reduction map is of the same quality as sixteen samples per pixel (and is sometimes better).

Table 4.3: Procedural Reduction Map Rendering Times: rendering costs for each of the 300 frame sequences. The average number of texture calls per frame in the procedural reduction map are listed in "Texture Calls (PRM)". The average number of texture calls per pixel in the 16 samples per pixel case are listed in "Texture Calls (16-Sample)". The timings for rendering the associated texture in the rotation animation is listed for "1-Sample" per pixel, "16-Samples" per pixel, and the "PRM" (which uses 1 sample per pixel.) Note that all models are have 10K triangles except the dragon, which is has 20K triangles. Rendering times are in (minutes:seconds).

Texture	Number of Texture Calls		1-Sample	16-Samples	PRM
	PRM	16-Samples			
Feline	6,392.7	39,868.6	5:01	13:16	6:20
Dragon	6,295.1	40,115.3	8:25	16:16	9:33
Stone Bunny	12,708.3	63,171.2	5:25	16:29	6:35
Bump Mapping	6,282.6	63,172.5	5:32	11:41	6:08
Igea	16,269.4	66,388.4	5:36	16:47	7:09

Table 4.4: Procedural Reduction Map Overhead. Cost of ray-object intersection tests and the procedural reduction map algorithm (all timings are in seconds.) Each row is a model with either 2 or 4 basis reflectance functions. The costs of texture computations has effectively been eliminated. Timings are for rendering 1,000,000 pixels. The second to last column is the additional cost of running the procedural reduction map algorithm. The last column is ratio of 1-Sample over PRM and demonstrates the algorithm runs at a little over half the speed of 1-Sample. The PRM for these examples does not make use of the normal distribution map.

Model	k	No. Triangles	1-Sample	16-Samples	PRM	Algorithm	Ratio
Igea	2	10,000	15.6	117.7	28.6	13.0	0.545
Igea	4	10,000	15.6	117.7	28.8	13.2	0.542
Dragon	2	20,000	24.1	186.6	42.1	18.0	0.572
Dragon	4	20,000	24.1	186.6	42.5	18.4	0.567

Therefore, it is often a savings in rays cast since the algorithm requires only one ray per pixel. However, the table also lists the number of texture calls. For many surfaces and associated textures, our algorithm has superior performance in the number of texture calls. Notice that our PRM technique is faster than 16 samples by a factor of 2 to 3, and produces comparable or higher quality results (see video).

The speed overhead of running the procedural reduction map algorithm is demonstrated in Table 4.4. To calculate this, the procedural texture returns only white with no computations, effectively eliminating the cost of texture calls from the timings. This leaves two costs: the ray/object intersection tests and the cost of running the procedural reduction map algorithm per 1,000,000 pixels (calculated by subtracting the timing for 1-Sample from the timing for the PRM.) To better display the cost of running the procedural reduction map algorithm, we use two models with different numbers of triangles as well as for each model using two different numbers (k) of basis reflectance functions. The normal distribution map version was not included in this since it is a blend of the procedural reduction map and regular super-sampling methods.

The animated scene with many procedural textures (shown in Figure 1.1 and in the accompanying video) has 840 frames and took 10.5 hours to render at 16 samples per pixel, while it took only 3.0 hours to render using procedural reduction maps (using 1 sample per pixel). Note that the 16 samples per pixel version still aliased at this rate. No minification aliasing can be seen in the PRM version of this sequence. Furthermore, over the 840 frames, the 16 samples per pixel version made over 280 million texture calls, while the procedural reduction map only made 60 million.

4.6 Conclusions

The procedural reduction map provides an automatic method of anti-aliasing most procedural textures. In addition to handling simple color variation, this approach also anti-aliases reflectance variations due to specular highlights. This algorithm is a general tool that will relieve the procedural texture author from the difficult task of writing code to anti-alias her textures.

The limitations to the system have been explained in detail throughout the chapter, but were expressly delineated in section 4.2. Removing, or at least diminishing, these assumptions are discussed in detail in the next section.

Although the initial analysis for a given texture is costly, this cost is amortized by using the procedural reduction map repeatedly in many frames of an animated sequence. Furthermore, a procedural texture is often created much earlier than it is used. In other words, a texture might be created one day, but its final usage might occur months or even years later. Therefore, the creation time is not truly a disadvantage in many cases.

The algorithm is also robust in most cases with regards to violations of the assumptions. The only time the procedural reduction map can fail is during its creation. If any failed assumption results in an inability to detect equivalent reflectance functions, then the number of identified basis reflectance functions can increase arbitrarily.[4] Otherwise, slight, and in some cases, major, violations to the assumptions still result in an anti-aliased version of the procedural texture, albeit with degraded visual fidelity.

4.7 Future Work

There are a few limitations of our system that should be investigated in the future. Assumptions 2-7 could all possibly be removed and/or reduced in severity. Furthermore, the Integration Problem was never handled. This will be alleviated in the next chapter.

In the future, it would be worthwhile to look into slightly less automated anti-aliasing systems that could reduce and/or eliminate some of the assumptions listed early in the chapter. Another important aspect would be to move procedural reduction maps onto the graphics processing unit, which will occur in chapter 6. The rest of this section is a brief discussion of some of those possibilities.

4.7.1 Removing the Distant Light Assumption

Although this assumption could be removed, it probably has little visual impact due to the minification of the texture during procedural reduction map usage. However, future work

[4]The system detects this by placing a bound on the number of basis reflectance functions and fails if the number exceeds that.

might be able to remove this assumption, which would not only help procedural reduction maps, but also MIP-Maps.

4.7.2 Removing the Light Sampling Assumption

Future research could look into how to remove this assumption. This involves two aspects. First, the reflectance function detection must be taken into account. Second, rendering such types of reflectance functions must also be examined.

The first part will be handled in chapter 6, but the second will not. However, one might be able to use current methods for anti-aliasing environment maps to handle this situation.

An interesting side effect of removing this assumption would be to use procedural reduction maps for global illumination. Secondary rays hitting a PRM object could give a better estimate of the color for a large region, allowing for fast radiance transfers.

4.7.3 Removing the Specular Lobe Assumption

Forcing the specular lobe to be in a specific place limits the types of reflectance functions available to anti-aliased. However, there is a simple method to alleviate this assumption, although it involves a little author intervention. Chapter 6 will discuss how to do this.

4.7.4 Removing the Isotropic Reflectance Function Assumption

This assumption is probably the most difficult to overcome. Even if the procedural texture could tell what its reflectance function is, integration of these high-dimensional reflectance functions would be very difficult. Future work might involve simplifying different aspects of how those type of reflectance functions work when minified, perhaps allowing for a tractable solution.

4.7.5 Removing the Non-Deformable Assumption

The easiest assumption to remove in certain cases is the Non-Deformable Model Assumption. In the cases of off-line rendering, when a model deforms, the simple, and theoretically sound, approach would be to rebuild the surface normal distributions for the affected areas. This would require keeping track of all original sample points on the surface, along with their original and perturbed (shader) normals. When points are moved according to the

deformation, those points are flagged and after the deformation the procedural reduction map's normal distribution map is recalculated for those points (and all texel's above it in the pyramid). Note that this would be an off-line process, but it still should be fast enough to warrant implementing.[5]

4.7.6 Removing the Time Invariance Assumption

Allowing a procedural texture to vary in time might be fine if the texture loops. Since time moves linearly, the procedural reduction map might be applied at each time step throughout the entire loop. This series of procedural reduction maps could then be combined into one large one. Future research, though, would be required in how to compress all of that data, which would be quite large. Other research might look into the possibility of non-looping textures with added assumptions about the time-variance.

4.7.7 Removing the Displacement-Free Assumption

Current methods for anti-aliasing displacement maps should be able to work concurrently with procedural reduction maps: one working with the geometry, the other with the reflectance function. However, this should be investigated further to hopefully eliminate this assumption.

[5]This also assumes that the deformed model did not "swim" through the procedural texture.

CHAPTER V

REFLECTANCE FUNCTION INTEGRATION

The Integration Problem listed in section 4.1 was never truly addressed in the preceding chapters. However, to achieve better visual fidelity to the correct reconstruction, better approximations to the integrated reflectance function are required. This chapter presents a method for achieving good approximations to the reflectance function integration.

5.1 Background Theory

First, a little background is necessary for a full appreciation of the problem. This section will describe the problems inherent in integrating reflectance functions: high dimensionality and reflectance function integration aliasing. Most of the example figures in this chapter are in two dimensions, but what they represent can easily be extrapolated to three. Also note that all items in the figures marked in red are constant within the corresponding figure.

5.1.1 Instances of Reflectance Function Integration

Before describing the difficulties, a little review of what reflectance function integration is and when it occurs is in order. Recall that the reflectance equation (equation 1.2) is:

$$L(x, \omega_o) = \int_{\omega_i \epsilon \Omega} L(x, \omega_i) f_r(\omega_i, \omega_o) \cos\theta_i d\omega_i \tag{5.1}$$

Reflectance function integration involves evaluating this equation as shown in figure 5.1.

Accurate rendering requires reflectance function integration under several circumstances. Another case is shown in figure 5.1 through figure 5.3. All of the following involve only one reflectance function.

The first such example is when the incoming light source is not a point light, but an area light. Traditional methods often employ point sampling, but more accurate would be to integrate over the reflectance function with regard to the area covered by the light source.

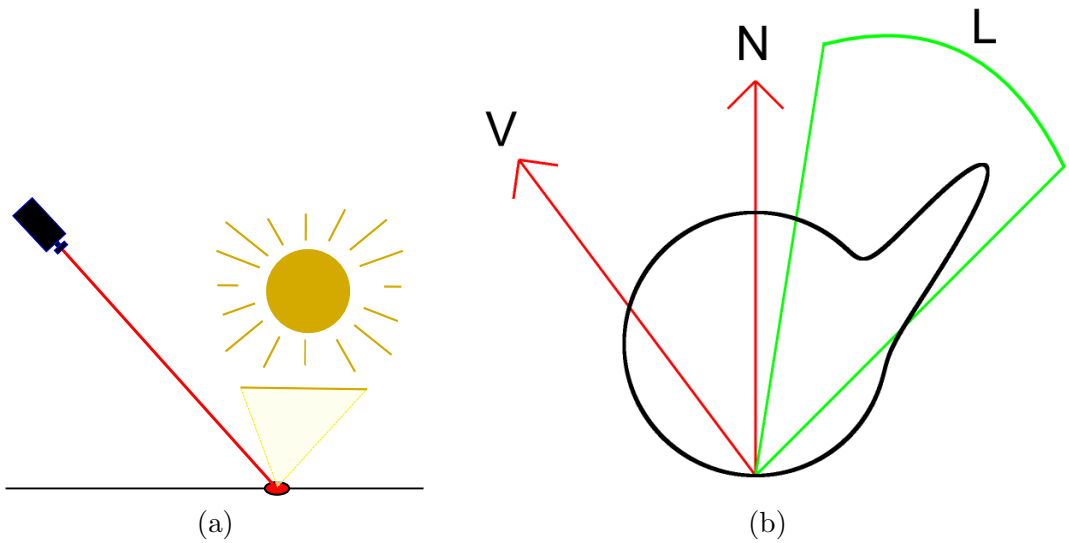


Figure 5.1: Required Reflectance Function Integration: Area Light Source, Point Surface. (a) Near area light source (yellow), distant (nearly constant) viewer (red line). (b) V, the outgoing light direction (ω_o) and the normal, N, are constant. The black curve represents the incoming light directions (ω_i). For a given direction, d, the distance of the black curve is equivalent to the intensity of the outgoing light in direction V if the incoming light is from direction d. The portion of the reflectance function that must be integrated is shown by the green curve (the area light source).

See figure 5.1. The rest of the examples will use a point light source, but they can all be applied to area light sources as well.

The second example involves a single point light source, but integrated over an area of the planar surface. As seen in figure 5.2, if the light and/or view is close enough to the surface,[1] then accurate rendering will require integration over an area of the reflectance function.

The last example relates to the previous one. Consider the same planar region as in the last example. If we make the area small enough, point sampling the reflectance function would be enough. However, if we kept the same small area but moved it onto a much more rough surface, then this would no longer be the case. Figure 5.3 demonstrates the case when reflectance function instantiation varies enough to warrant reflectance function integration. This often occurs with bump maps.

From these cases, we see that integrating the reflectance function (equation 5.1) results

[1] Obviously, for minification cases, the view will not be close enough for this to be the case.

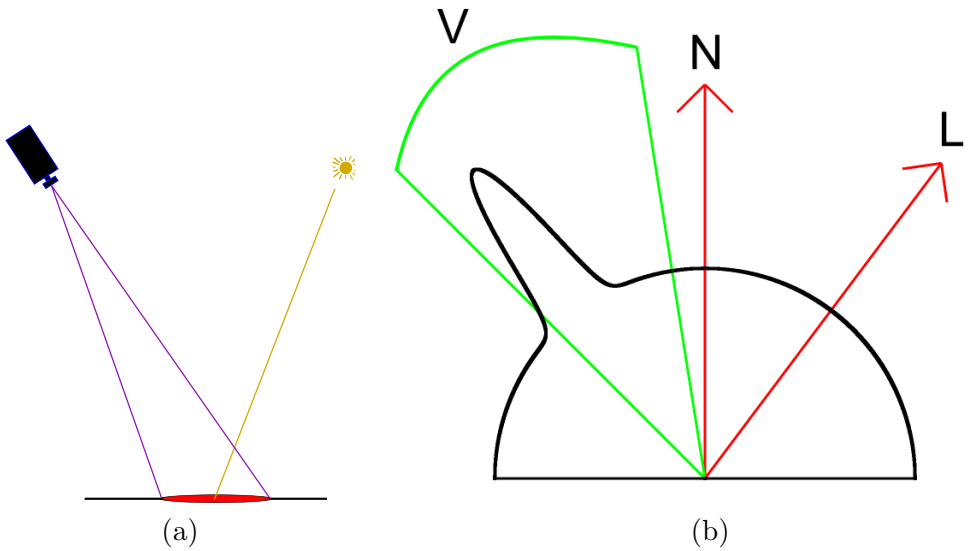


Figure 5.2: Required Reflectance Function Integration: Point Light Source, Surface Area. (a) Large surface area (red ellipse) seen by pixel, distant (near constant) light source (yellow line). (b) L, the incoming light direction (ω_i) and the normal, N, are constant. The black curve is the intensity of the outgoing light in that direction. The green portion of the reflectance function is the outgoing light directions that are in the pixel's view. The integration is therefore over the green portion of the curve.

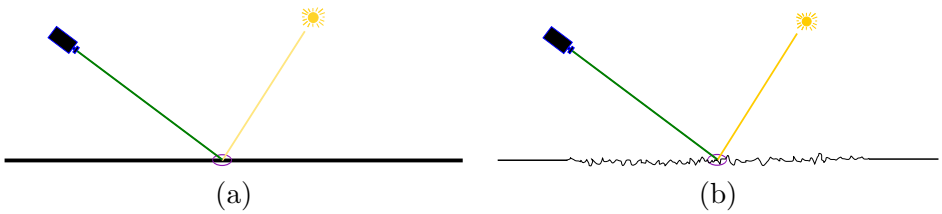


Figure 5.3: Required Reflectance Function Integration: Varying Normals. (a) The planar surface is flat with constant incoming (yellow) and outgoing (green) light directions. Point sampling the reflectance function is sufficient here. (b) The same situation, except the surface is rough, with varying surface normals. An integration of the reflectance function is required.

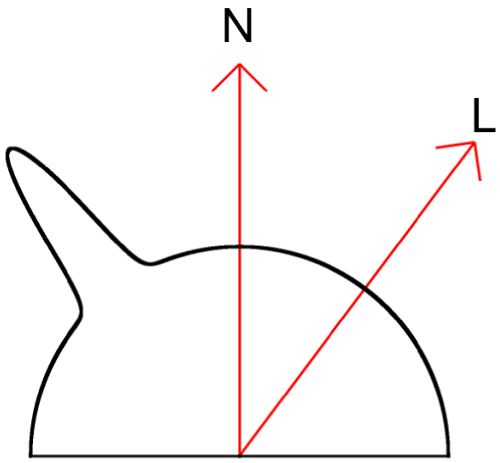


Figure 5.4: Traditional BRDF Description. The incoming light direction, L (or ω_i), and the normal, N, are constant. The black curve represents the intensity of the outgoing light in the corresponding direction.

in:

$$\Phi(x) = \int_{x \epsilon S} \int_{\omega_i(x) \epsilon \Omega_x} L(x, \omega_i(x)) f_r(x, \omega_i(x), \omega_o(x)) cos(\theta_i(x)) d\omega_i(x) dx \tag{5.2}$$

where Φ is the flux at the pixel sensor, and x is the surface area seen by the sensor.

5.1.2 High Dimensionality

As discussed in chapter 4, procedural textures can have high dimensional reflectance functions. Integration of the reflectance function must be done over several dimensions to cover all situations. Since dimensionality was discussed in great detail in the previous chapter, the discussion in this section will pertain only to isotropic BRDFs. The difficulty demonstrated with these reflectance functions will obviously be exacerbated with higher dimensioned functions.

Figure 5.4 demonstrates traditional reflectance functions. These representations reflect what happens in reality: incoming light determines all outgoing light.

The problem, however, with this representation is that it does not help us with anti-aliasing reflectance functions during minification. Consider figure 5.5. The outgoing light remains constant, but the incoming light does not. This means that the integration of a reflectance function becomes constant with regard to the outgoing light direction ω_o:

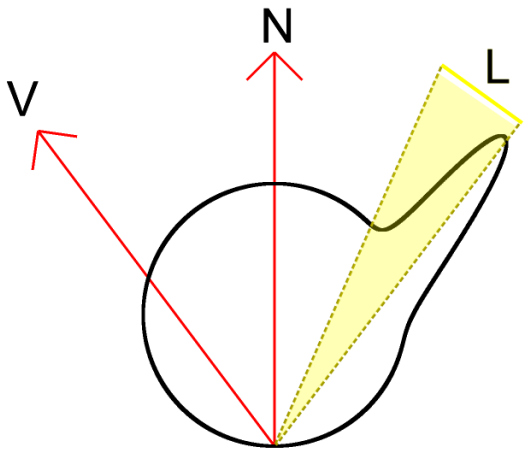


Figure 5.5: Actual Integration During Texture Minification. The planar surface has a constant normal, N, but the outgoing light direction, V, is also constant because texture minification only occurs when the viewing source is distant to the object. Therefore, the only thing varying is the incoming light. Notice the difference in which parameters are constant from figure 5.4.

$$\Phi(x) = \int_{x \epsilon S} \int_{\omega_i(x) \epsilon \Omega_x} L(x, \omega_i(x)) f_r(x, \omega_i(x), \omega_o) cos(\theta_i(x)) d\omega_i(x) dx \tag{5.3}$$

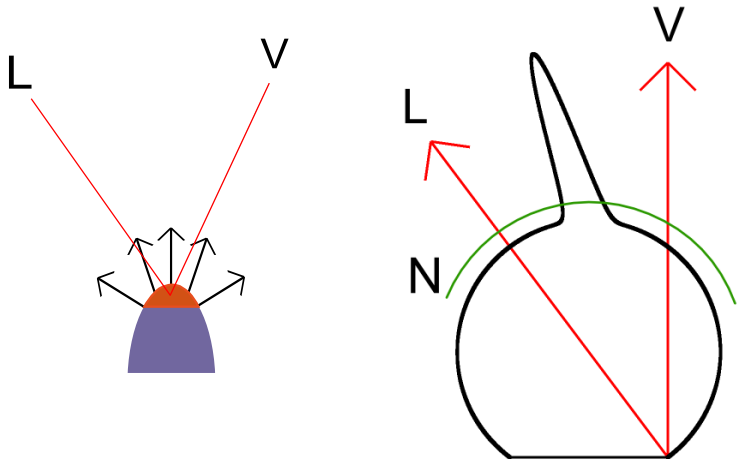
where ω_o is constant for all x. Current standards of integration do not account for this, and are left with requiring many point samples during run-time.

However, by keeping the Distant Light Assumption, $L(x, \omega_i)$ is zero everywhere in the integral except when ω_i points toward the light. This is equivalent to having a Dirac delta distribution in the direction of the light and everywhere else be zero. Therefore, we can transform equation 5.3 into the new integrated reflectance function:

$$\Phi(x) = \int_{x \epsilon S} \int_{\omega_i \epsilon \Omega_x} L(x, \omega_i) f_r(x, \omega_i, \omega_o) cos(\theta_i) d\omega_i dx \tag{5.4}$$

We now draw out the constants within the integral:

$$\Phi(x) = L(x, \omega_i) \int_{x \epsilon S} f_r(x, \omega_i, \omega_o) cos(\theta_i) dx \tag{5.5}$$



(a) (b)

Figure 5.6: Reflectance Function Instantiation. (a) Even for a small area, if it has a wide enough normal distribution, reflectance function integration is required. Here, both the incoming and outgoing light directions (L and V) are both constant. However, the normals are not constant, and thus (b) the reflectance function must be integrated over the normals shown in green.

Even though the equation has been simplified, there is still the problem of reflectance function instantiation. Recall that a rough surface that, while it has only one reflectance function, will not have a single (ω_i, ω_o) but a distribution of them due to the varying shader normals (figure 5.6). This makes the problem more difficult because we now have to work with a distribution of normals. Our two dimensions (incoming and outgoing light directions) has now become three! Moving from the two dimensional cases we've been examining to three dimensions, the pre-computation of reflectance function integration would require four dimensions for the isotropic BRDF. Equation 5.5 can now be described in terms of the reflectance function instantiation:

$$\Phi(x) = L(x, \omega_i) \int_{x \epsilon S} f_r(N(x), \omega_i, \omega_o)(N(x) \cdot \omega_i) dx \tag{5.6}$$

where $N(x)$ is the normal at point x. The distribution of normals therefore determines the evaluation of the integral.

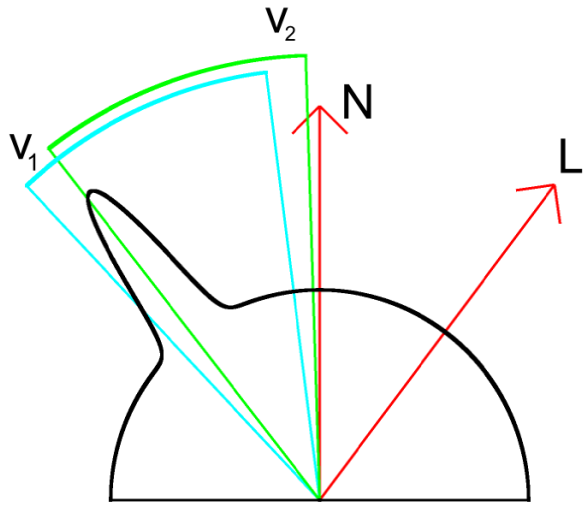


Figure 5.7: Reflectance Function Aliasing. A traditional BRDF is shown with constant N and L. The blue and green arcs represent different samplings of a reflectance function integration. The difference in angle between the green and the blue is less than four degrees. The difference in the resulting pixel intensity is four percent, enough to make a significant difference to the human visual system.

5.1.3 Reflectance Function Integration Aliasing

The last difficulty in integrating reflectance functions is aliasing. Figure 5.7 demonstrates aliasing that can occur during poor reflectance function integration. Furthermore, the figure demonstrates that small differences in the integration can lead to dramatically different results.

5.2 Normal-centric Bidirectional Reflectance Distribution Function

Pre-computed integration of reflectance functions appear to be too expensive for normal rendering. However, by changing the parameters of the reflectance function, we are able to have an intuitive and relatively inexpensive evaluation of reflectance function integrations. The name of this method is called the *Normal-centric Bidirectional Reflectance Distribution Function*, or *NBRDF*.

This method assumes that the incoming and outgoing light directions are nearly constant.[2] Since this method is for anti-aliasing minification artifacts, the outgoing light direction assumption is, in practicality, given to us. Since it is a common assumption (i.e., MIP-Maps), we again use the Distant Light Assumption to deal with the incoming light source.

For now, also assume the reflectance function is no more complex than an isotropic BRDF.

5.2.1 (2-D) Algorithm Overview

This method does not reduce the dimensionality of the reflectance function integration problem, but rather transforms the problem to fit the needs of minification anti-aliased rendering. This section gives an overview of the process. For clear exposition, the discussion in this section will be in two dimensions.

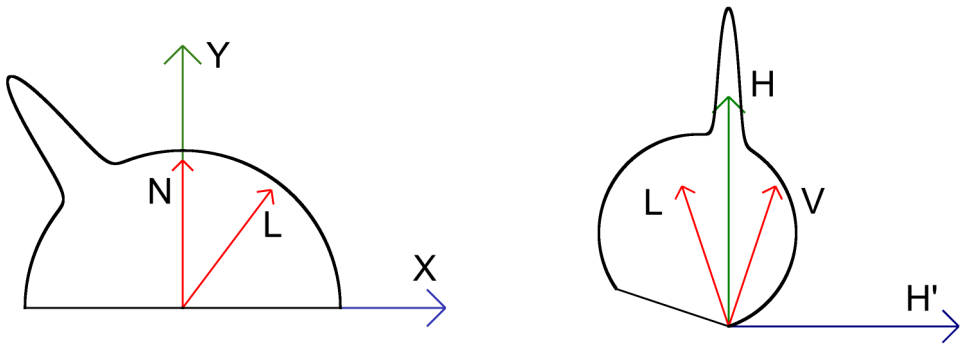
The first step transforms the coordinate axes of traditional reflectance functions to a normal-centric set of axes. Figure 5.8 demonstrates the transformation from a (single normal, single incoming light direction, and multiple outgoing light directions) coordinate system to a (single incoming light direction, single outgoing light direction, and multiple normals) coordinate system. This, in effect, transforms the integration problem into equation 5.7.

By observing the fact that ω_i and ω_o do not change, we can describe $N(x)$ in terms of ω_i and ω_o. By creating a coordinate system from the light directions, we transform $N(x)$ to $N'(x)$. For a specific instance of the reflectance function integral, equation 5.7 becomes:

$$\Phi(x) = L(x, \omega_i) \int_{x \epsilon S} f_r^\star(N'(x))(N'(x) \cdot \omega_i) dx \tag{5.7}$$

where $f_r^\star(N'(x)) = f_r(N(x), \omega_i, \omega_o)$ for the specific constants ω_i and ω_o. We call $f_r^\star$ a *reflectance slice*. Note that the reflectance slice is for a specific, constant, ω_i and ω_o. Although it might be counterintuitive, only a few reflectance slices are required to reconstruct an integrated reflectance function, even one with a high specular term, as shown in

[2] Which is equivalent to saying the viewing and lighting sources are distant with respect to the surface.



(a) (b)

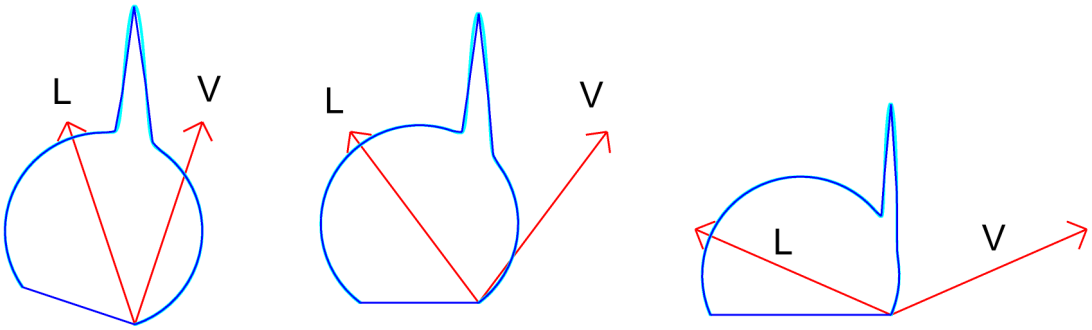
Figure 5.8: 2-D Normal-centric Reflectance Function Transformation. (a) Traditional BRDF with constant N and L. The coordinate system is $Y(=N)$, and X, which is orthogonal to Y. (b) Normal-centric coordinate system consists of H the half-vector between L and V, and H' the vector orthogonal to H.

figure 5.9. The summation of all reflectance slices is equivalent to the original reflectance function, $f_r(x, \omega_i(x), \omega_o(x))$.

The summation of all reflectance slices fully describe the integrated reflectance function in tabulated form. Upon first inspection, it might appear that there are two independent dimensions: the incoming light direction and the outgoing light direction. However, this is not the case, there is only dimension: the angle between the two directions. The incoming and outgoing light directions, ω_i and ω_o, describe the coordinate system of $N'(x)$. Therefore, if we rotate the light directions exactly the same, then the coordinate system has only rotated, and equation 5.7 remains the same. Another way to look at it is that wherever the incoming and outgoing light directions are, the two can be rotated so that the incoming light direction matches the "canonical" incoming light direction, as shown in figure 5.10. The only difference is the angle between the two directions.

The range of reflectance slices is from 0 to π radians. Figure 5.11 demonstrates a series of reflectance paths, collectively known as the *reflectance calender* (named because the 3D diffuse version resembles a lunar calender).

The reflectance calender describes in tabular form a low-level integration of the entire reflectance function. Each sample represents a distribution of normals. Therefore, rendering



(a) (b) (c)

Figure 5.9: 2-D Discrete Version of three Reflectance Slices. Three different reflectance slices are shown, each with a specific constant L and V. The light blue is the actual function and dark blue is the segmented version. There were 64 segments used in each reflectance slice.

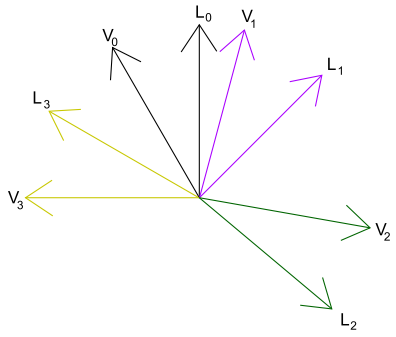


Figure 5.10: 2-D Changing Coordinate Systems. Each color pair of vectors represents a different incoming and outgoing light directions. However, they can all be rotated to match the "canonical form," the black pair of vectors, L_0 and V_0. Therefore, all of these cases are equivalent.

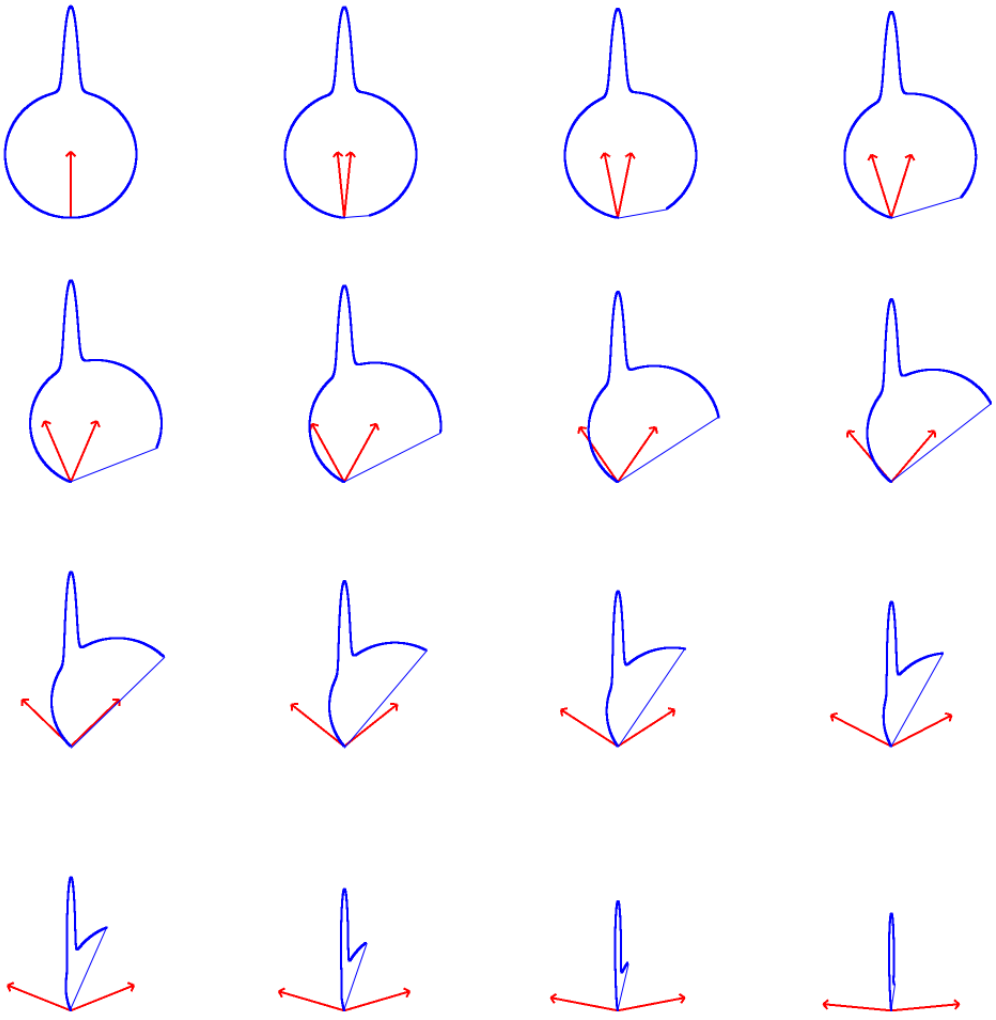


Figure 5.11: 2-D Reflectance Calender. For each reflectance slice, the red arrow on the left is the outgoing light direction and the right arrow is the incoming light direction. There is a sharp cutoff when the normal direction is facing away from the viewing direction.

a distribution of normals is simply the average of the associated samples within a reflectance slice.

For fast rendering, however, footprint assembly is not a good idea. Therefore, a pre-integrated hierarchy of tables is created based on the exponent. For diffuse and glossy reflectance functions, a pyramid (like MIP-Maps) suffices.

For specular components with high exponents, though, more precision is required (see figure 5.7). Instead of a pyramid, a set of layers, all having the same resolution, are used. This is similar to Lefebvre and Hoppe's Gaussian stack [63]. Each progressively higher level has texels that contain wider distributions of normals per texel.

The next two sections discuss some implementation details for creating and rendering the NBRDF, as well as any interesting facets in moving to three dimensions.

5.2.2 Creation

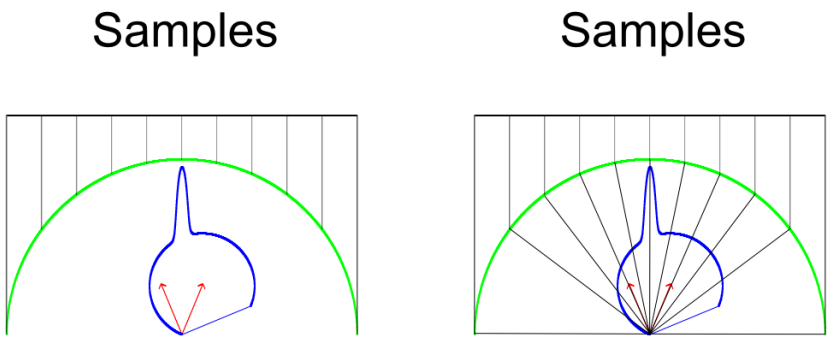
Creating a NBRDF requires three inputs: the reflectance function, base size of the hierarchy, and the type of hierarchy. The base size of the hierarchy determines the number of samples in each reflectance slice. The type of hierarchy can either be a pyramid or the layered hierarchical scheme. Because we will later use texturing hardware to implement reflectance slices, we will sometimes refer to samples in a slice as "texels."

The move to three dimensions is straightforward for most aspects. Since the reflectance function is isotropic, the space of different reflectance paths is still one dimensional. Furthermore, the isotropic reflectance function combined with reciprocity from the nature of light[3] means that there is symmetry along the incoming light direction.

Each texel represents an integration of the reflectance function for a specific incoming and outgoing light directions over a distribution of normals. In two dimensions, the storage of such information is a simple array. In three dimensions, the distribution of normals is over a sphere, and not as intuitively stored.

Figure 5.12 shows how to creates texels for a reflectance slice. By viewing the reflectance

[3]Not all reflectance functions obey this, however, the difference between the two values is often insignificant. For our purposes, it can be ignored. The symmetry is only an observation of a reflectance slice.



(a) (b)

Figure 5.12: 2-D Creation of Samples for a Reflectance Slice. A reflectance slice is shown in blue with the incoming (right) and outgoing (left) directions in red. (a) The samples are projected onto a hemisphere covering the reflectance function. (b) The portions of the hemisphere are projected onto the reflectance function. The resulting values correspond to the reflectance function if the normal is in that direction.

slice from above, each texel represents a distribution of normals centered around the projection of the texel onto the sphere.

Since the reflectance function is not more complex than an isotropic BRDF, then no light will transmit through the surface. This means that the normals that are on the opposite side of the hemisphere from the incoming light will be black. Furthermore, since the outgoing light also cannot be transmitted through the surface, the hemisphere on the opposite side of the outgoing light must also be black. Therefore, by centering the projection of texels on the vector halfway in-between the incoming and outgoing light directions, we achieve maximum resolution where most specular highlights occur and less resolution at the light silhouette.[4]

The final consideration is the normal distribution represented by a texel. For a pyramid hierarchy, the base level is super-sampled and then the result is pulled up the pyramid. Texels that represent *false normals*, that is texel projections that lie outside the sphere of normals, do not contribute to the scheme at all, and therefore there is no need for pushing the values back down the pyramid.

[4]The light silhouette is the surface's silhouette as seen by the light source.

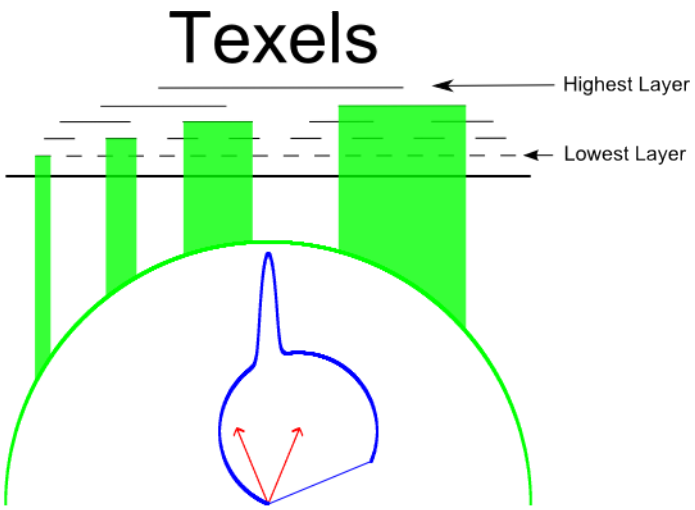


Figure 5.13: 2-D Layered Reflectance Calender Texel Calculation. The higher in the pyramid, the larger the distribution of normals covered per texel.

For the layered hierarchy, a texel's normal distribution size is based on how high in the hierarchy the texel is located. The higher the layer, the larger the distribution size, as shown in figure 5.13. Each texel is calculated by a Gaussian blend of super-sampling the corresponding distribution of normals. An example of the lower level of a NBRDF is shown in figure 5.14.

5.2.3 Rendering

During rendering, the NBRDF is called with four parameters: the incoming and outgoing light directions and the normal distribution (the average normal and the size of the distribution). The first step compares the distribution size to the lowest level of the hierarchy. If the size is less than the lowest level's representation, then the original is used, otherwise the algorithm continues. The second step computes which two reflectance slices are the closest to matching the incoming and outgoing light directions. The third step computes the (u, v) coordinates for the calender slice. The next step calculates the intensity for each calender slice based on the distribution size. The final step linearly interpolates the answers from the two reflectance slices.

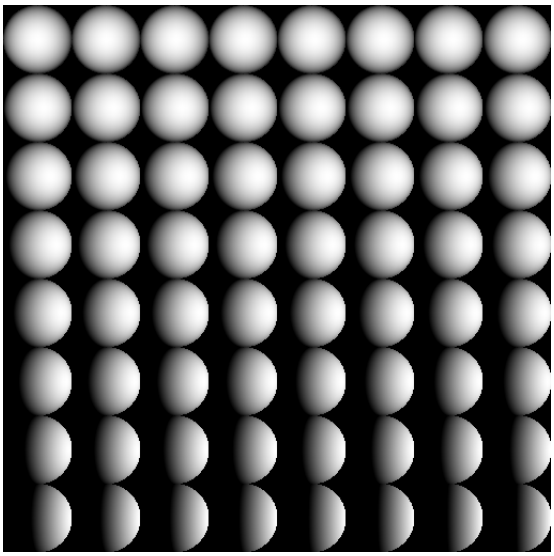


Figure 5.14: 3-D Lowest Level of a Diffuse Phong Reflectance Calender. The angle between the incoming and outgoing light directions starts at zero (upper left reflectance slice) and goes all the way (left to right, top down) to just shy of π radians. Each reflectance slice has the incoming light direction in the positive x axis, and the outgoing light direction in the negative x-axis. The reflectance calender shown does not differentiate when the normal is back-facing, so there is only a sliver of the lower reflectance slices that are actually used (since no lookup will be based on a back-facing normal.) Note that a reflectance slice is symmetric about the x-axis if the reflectance model obeys Helmholtz Reciprocity. The name comes from its similarity to the lunar calender.

The NBRDF first computes whether to use an integrated form or not. Similar to procedural reduction maps, there are three options: use the original reflectance function, use the NBRDF, or use a linear blend of the two. If the distribution size of the pixel, D_p, is at least as large as the associated distribution size of the lowest level texel, D_t, then use the NBRDF. If D_p is less than the $\frac{D_t}{k}$, where k is the super-sampling rate of the NBRDF low-level texels, then use the original reflectance function. Otherwise, use a linear blend of the two.

The second step is straightforward. By calculating the angle between the incoming and outgoing light directions, the two reflectance slices that are closest to the actual angle are chosen. Weights summing to one are assigned to each reflectance slice according to their proximity to the actual angle.

The third step computes the (u, v) coordinate for a reflectance slice (see figure 5.15.) Note that the (u, v) coordinate is the same for all reflectance slices if we carefully define the coordinate system.[5] Once we have the axes for the reflectance calender, projecting the normal into that basis gives the (u, v) coordinates.

For both types of hierarchical schemes, the fourth step runs just like a MIP-Map, with the normal distribution size serving as the variable to determine how high to go within the pyramid. Trilinear interpolation is used to determine the answer. The only difference is that for the layered hierarchy, all distribution sizes above the equivalent size of the highest level in the hierarchy are clamped. Since the function is high frequency, this cutoff means minimal difference to the actual answer.

The final step weights the answers from the two reflectance slices appropriately for a final integration of the reflectance function. Note that this requires two MIP-Map look-ups and two layered hierarchical look-ups (which are equivalent to MIP-Maps in speed.) Therefore, four equivalent MIP-Map look-ups are required. However, it should be noted that if multiple reflectance functions are layered on the same portion of the surface, then

[5]This is technically not true, since the reflectance slice representing identical incoming and outgoing light directions cannot give us enough information to determine a distinct set of axes. However, since the function is isotropic, this does not matter.

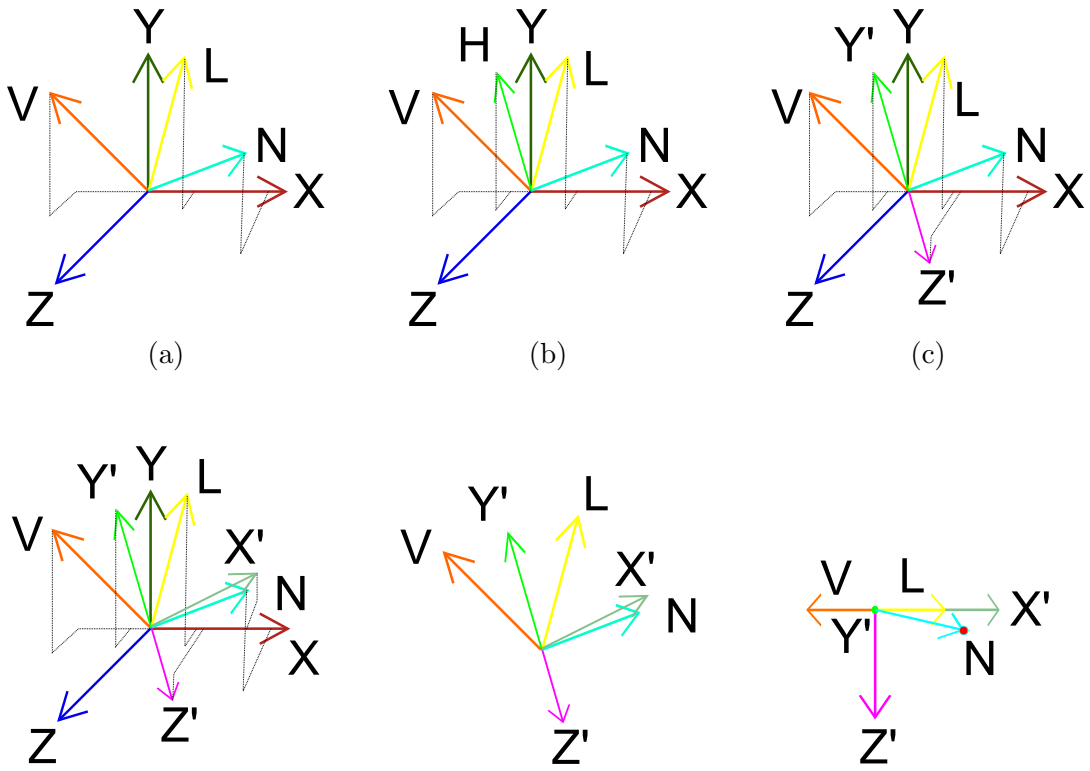
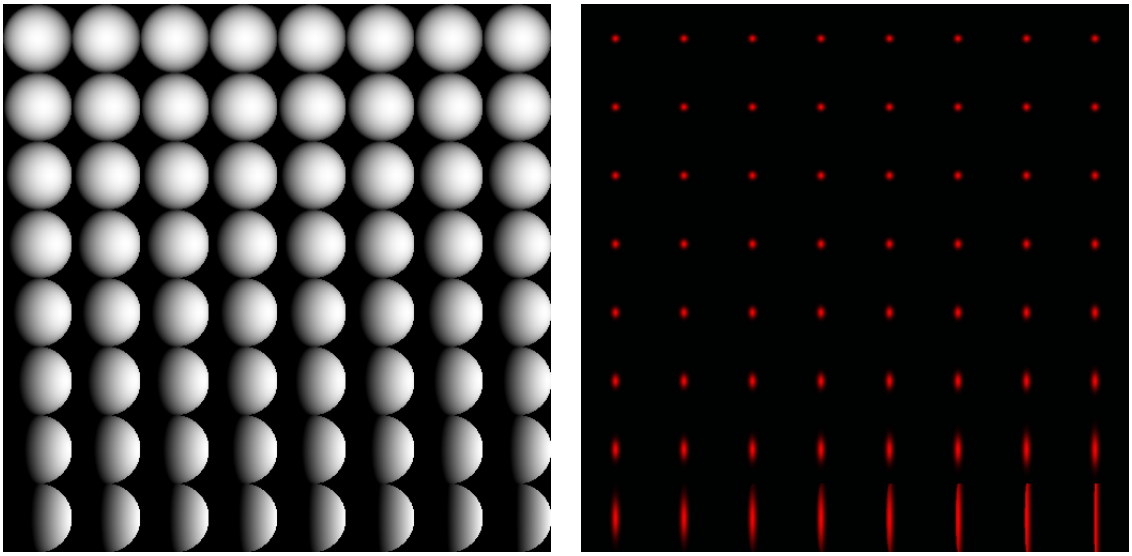


Figure 5.15: 3-D Calculating (u, v) Reflectance Slice Coordinates.
(a) We start with an XYZ (red, green, blue) coordinate system. We have incoming (yellow) and outgoing (orange) light directions and a normal. The dotted lines help visualize where in space the vectors are.
(b) We compute the H half-vector between V and L.
(c) We say H is our new Y' axis, and compute the new Z' axis as $Z' = L \times V$.
(d) We compute the new X axis: $X' = Y' \times Z'$.
(e) Our new coordinate system, $X'Y'Z'$.
(f) A bird's eye of view of the coordinate system, with Y' coming out of the page. The (u, v) coordinates of N are the (x', z') coordinates of N.



(a) (b)

Figure 5.16: Phong Reflectance Calender. (a) Diffuse Phong. (b) Specular Phong with an exponent of 50.

these could be done concurrently.[6]

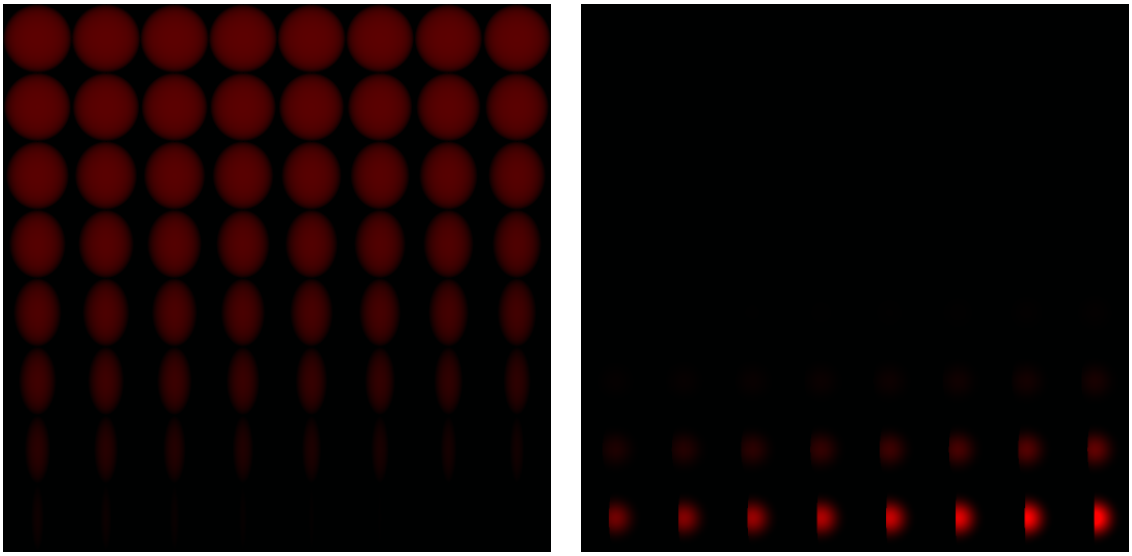
5.3 Results

NBRDFs were tested on a dual-processor Intel Xeon, 3.2 GHz machine with 2 GB of memory. The NBRDF creation program was independently written. We note that future implementations of the creation program will probably need to be written to interface with the original procedural texture (i.e., if the procedural texture is a RenderMan procedural texture, then the creation program should interface with that kind of procedural texture.) Rendering was done with rasterization hardware, although there should be no problem using a ray tracer.

Figure 5.16 demonstrates the base level of different Phong model reflectance functions and figure 5.17 does the same for the Ashikhmin model. For the highly specular components, a layered hierarchy was used. For the diffuse and glossy components, a pyramid was sufficient. Note that these are not the only type of reflectance function models possible. They were merely chosen for the ubiquity (Phong) and their expressiveness (Ashikhmin).

Table 5.1 lists the timing results for NBRDF creation as well as the storage costs. Note

[6]We will return to this in the next chapter.



(a) (b)

Figure 5.17: Isotropic Ashikhmin Reflectance Calender. (a) Diffuse portion of Ashikhmin BRDF. (b) Fresnel, isotropic specular portion of Ashikhmin. $R_d = R_s = 0$ and $n_u = n_v = 10$.

that the creation time for each component is one of two constants, depending on whether it is diffuse/glossy or highly specular. The high creation times for specular components is due to the sampling of the surface normal distribution at each texel in the hierarchy. Furthermore, due to reuse (i.e., the diffuse component), creation times will decrease over time. For example, if a library stores common reflectance function components, such as diffuse, glossy (Phong exponent of 10), and specular (Phong exponent of 100), then any time a procedural reduction map requests those as basis reflectance functions, then no creation is required.

All NBRDFs fell into one of two categories for storage cost. If the NBRDF was diffuse or glossy, then it became a pyramid, otherwise it became an 8-level hierarchy. The base of each hierarchy was made up of 64 reflectance slices, with each reflectance slice having 64×64 texels. Therefore, the base level was 512×512 texels. One component could fit in one byte, so one the base level of one component was 0.25 MB.

Pyramid NBRDFs (diffuse and glossy) totaled 0.33 MB per component. Layered NBRDFs (highly specular) totaled 2 MB per component, due to 8 levels, each equivalent to the base level in size.

Table 5.1: NBRDF Creation Times and Storage Costs. Several types of Phong and Ashikhmin reflectance functions are shown with the corresponding exponent. The time to create each reflectance function is shown in the fourth column. The base dimensions of the lowest level of the NBRDF is 512×512. Note that this is required only once per reflectance function for all procedural textures. The size of the NBRDF is constant (0.3 MB) for all diffuse reflectance functions and constant for all specular reflectance functions (2 MB).

Model	Component	Exponent	Timing (m:s)	Storage (MB)
Phong	Diffuse	n/a	0:45	0.3
Phong	Glossy	10	0:45	2.0
Phong	Specular	50	27:29	2.0
Phong	Specular	100	27:29	2.0
Ashikhmin	Diffuse	n/a	0:48	0.3
Ashikhmin	Glossy	10	0:48	2.0
Ashikhmin	Specular	50	28:49	2.0
Ashikhmin	Specular	100	28:49	2.0

Figure 5.18 demonstrates the use of a NBRDF rendered on the GPU. More results will be shown in the next chapter.

No timings are given for rendering since it is done on the GPU. However, if the input given to the NBRDF is a (u, v) coordinate, an estimate of the texture footprint (i.e., size of the distribution), and the angle between the incoming and outgoing light directions, then it runs at half the speed of a MIP-Map (because it makes two MIP-Map calls.) However, if that input is not given to the NBRDF, then that must be calculated as well.

5.4 Conclusion

The Normal-centric Bidirectional Reflectance Distribution Function handles reflectance function integration well. It is relatively computationally inexpensive, allows for better visual fidelity during minification, and is fast during rendering. Aliasing artifacts are significantly reduced and/or eliminated. As will be shown in the next chapter, the NBRDF works well with procedural reduction maps, thereby increasing the visual fidelity of procedural reduction maps. Furthermore, through the use of NBRDFs, there is no longer a need for the multiple samples approximation (in screen space, section 3.3.2). This approach handles even specular integration well. Therefore, we can now handle the Integration Problem stated in section 4.1.

Figure 5.18: Procedural Reduction Map using the NBRDF. This procedural texture demonstrates a highly specular bump mapped bunny on the GPU. Through the use of the NBRDF, minimal aliasing occurs.

5.5 Future Work

This section discusses the ways in which the NBRDF could be furthered, allowing for better procedural reduction maps, both in procedural textures it can anti-alias, as well as the quality of that anti-aliasing.

5.5.1 Gaussian Mixture of Distributions

A texel in the NBRDF represents a Gaussian distribution of normals. However, for coarse geometry, it might make more sense to use a Gaussian mixture of normal distributions as Tan et al. did to have even better approximations of the portion of the reflectance function to integrate [98].

5.5.2 Better Representation

Currently, the representation of the NBRDF is a tabulated form. Future work should look into better representations of the NBRDF, such as spherical harmonics, spherical wavelets, factored BRDFs, etc. Such representations would not only help with storage, but possibly allow for reflectance functions of higher dimensionality, including anisotropic reflectance.

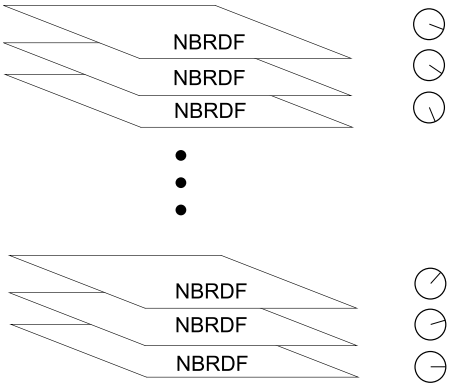


Figure 5.19: Anisotropic Reflectance Function Integration. Each NBRDF will represent one anisotropic tangent direction.

5.5.3 Higher Dimension of Reflectance Function

In the future, higher dimensional reflectance functions should be pursued to increase the procedural reduction map capability. For example, anisotropic BRDFs might be handled by having a stack of NBRDFs on top of each other, each one representing a different tangent axis (see Figure 5.19). However, these would also need to be combined, whether pre-computed and stored, or computed during run-time. The size of this type of NBRDF might be prohibitive, though, and compression is a matter of future research.

CHAPTER VI

TEXTURE AUTHORING AND GPU RENDERING

Fully automatic procedural reduction maps were introduced earlier in this thesis to anti-alias a wide range of procedural textures. Due to issues such as high dimensionality and reflectance function detection, simplifying assumptions had to be made. Some of these assumptions will not be true for some useful textures.

While fully automatic anti-aliasing of procedural textures is desirable, it should be clear that constraining the way procedural textures are authored might give us more information with regards to reflectance function detection. If these constraints are intelligent and intuitive, it will still be easy to author procedural textures.

This chapter serves the dual purpose of implementing a rasterization version of procedural reduction maps (moving the technique onto the *Graphics Processing Unit*, or *GPU*) as well as constraining the manner in which procedural textures are authored to reduce and/or eliminate as many assumptions that were discussed in chapter 4.

For the most part, the GPU implementation is the same as the ray tracing method. However, as differences arise, they will be mentioned. Most of the differences, though, will be a result of having extra reflectance function information about the procedural texture through the new way in which they are authored.

The next section details all the differences between GPU rasterization and a ray tracer. The following section explains the differences when semi-automatic methods are employed. The third section explains GPU implementation details. The final two sections are the results and conclusions.

6.1 Rasterization

The input for creating a procedural reduction map remains the same: a (u, v) parameterized object and a procedural texture. Since the creation process ignores the type of renderer and just calls the procedural texture directly, there is not really any change to procedural

reduction map creation when using a rasterizer.

Rendering, however, does involve some differences. The first such difference involves the texture footprint estimation. Since the renderer computes the footprint based on polygons, then if any given polygon's (u, v) coordinates all fall within the same patch boundary, then no footprint size estimation errors will occur. This eliminates the need for pixel super-sampling when (u, v) coordinates cross patch boundaries.

The other difference is handling the normal distribution. For some rasterization renderers, there can be no screen-space multiple samples approximation. For those, the NBRDF method must be employed. For off-line rasterization renderers, this is often not a problem and the original can be used, or a combination of the two, as desired.

If the the NBRDF method is employed, then there is no need to store the full Normal Distribution Map. The average normal and the distribution size are all that is required.

6.2 Authoring Guidelines for Procedural Reduction Maps

The goal of this section is to increase the usefulness of procedural reduction maps by adding some constraints to the method that procedural textures are authored. The assumptions from section 4.2 will be re-examined in light of added information. Specifically, those assumptions that were necessary due to reflectance function detection can be relaxed.

We are constraining the authoring process to eliminate the costly task of basis reflectance function identification during procedural reduction map creation. Furthermore, many BRDF implementations do not meet our requirements for shader normal determination (i.e., that the reflectance function returns black when the incoming and/or outgoing light direction is below the surface.) Therefore, the authoring constraints will help eliminate this problem as well. However, the authoring constraints also give us much more information about the procedural texture and we can remove three other assumptions as well: the Light Sampling Assumption, the Specular Lobe Assumption, and the Displacement-Free Assumption. (We also eliminate the reflectance detection problem of the Isotropic Reflectance Functions Assumption, but since we do not offer a way to render anything above isotropic reflectance functions, the assumption remains.)

6.2.1 Reflectance Function Modeling

The key to eliminating these assumptions is to recognize that there are few reflectance models in existence, and a significant amount of time in-between new ones appearing. All photo-realistic procedural textures use some kind of reflectance model. It may be a mixture of models, but at the end of much computation, a reflectance model is what is used to interact with the light within the scene.

Therefore, we provide reflectance function classes to shader authors for two reasons. First, they render the traditional reflectance function model they represent. For example, one class might represent all Phong BRDFs. Another might represent all Ashikhmin BRDFs. The other piece of functionality is that they will interface with the procedural reduction map algorithm.

The interface will perform two operations. First, an instantiation of a reflectance function model will take as input a list of basis reflectance functions and determine if it contains a new basis reflectance function or not. Second, the instantiation will return the appropriate basis weights for the corresponding basis reflectance functions in the list.

For example, consider two instantiations of the Phong BRDF. Both have diffuse and specular components. One has red diffuse and the other green. The first has a specular exponent of 10 and the other 100. If the list of basis reflectance functions was originally empty, then after passing the list to the first Phong BRDF, it would have two basis reflectance functions: a diffuse Phong BRDF, and a specular Phong BRDF with exponent 10. It would return weights corresponding to the diffuse coefficient and the specular coefficient. The list is then passed to the second Phong BRDF. After that, the list would contain three basis reflectance functions: a diffuse Phong BRDF, a specular Phong BRDF with exponent 10, and a specular Phong BRDF with exponent 100. The basis weights for that instantiation would correspond to the appropriate coefficients (the second basis weight would be 0.)

The parent class, *RFModel*, of all reflectance function models is listed in figure 6.1. It demonstrates the interface between the reflectance function models and the procedural textures.

```
class RFModel {
public:
    RFModel() {}
    virtual ~RFModel() {}

    // Determine the outgoing light intensity. (Regular rendering)
    virtual Color Intensity( const Vector &IncomingLight,
        const Vector &OutgoingLight, const Normal &N,
        float LightIntensity ) const = 0;

    // Add a basis reflectance function to the list if this
    //  instance is not already fully included in the list.
    virtual bool AddBasis( List &basis_list ) const = 0;

    // Determine the weights for each basis reflectance function
    //  according to this instance.
    virtual bool GetWeights( List &basis_list ) const = 0;

private:
};
```

Figure 6.1: The Parent Class of a Reflectance Function Model.

Because reflectance function models know what differentiates between scalings and non-scalings, the reflectance function models can delineate between two reflectance functions that sample the scene differently or even between two reflectance functions with multi-lobes. Therefore, we have eliminated the Light Sampling Assumption and the Specular Lobe Assumption.

An example demonstrating the elimination of the Specular Lobe Assumption is shown in figure 6.2. It shows a NBRDF of a multi-lobe reflectance function.

Note that the authoring of these classes (Ashikhmin, Phong, Ward, etc.) are done only once per reflectance function model. Different instantiations need not be re-written.

6.2.2 Procedural Texturing

We need the procedural reduction map to interface with the procedural texture as well, but we need access to the final reflectance function that is used at a sample point. Therefore, we also require all procedural textures to inherit from a parent texture, *ParentTexture*.

There are three aspects of a procedural texture that the procedural reduction map

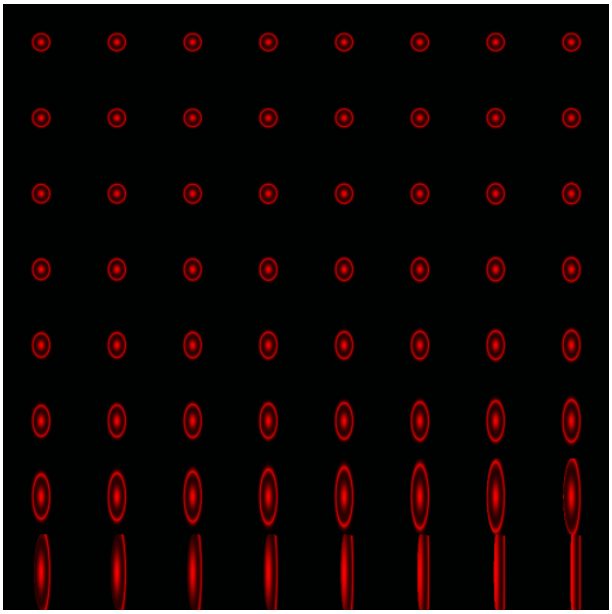


Figure 6.2: Varying Multi-lobed NBRDF. Only the specular portion is shown.

would benefit greatly knowing. First, the reflectance function model used at a specific point. Second, what the shader normal is. Third, the displacement of the point.

We allow the parent procedural texture to do nothing (i.e., return the input) for the last two functions. If a procedural texture wants to change the geometric normal, or perturb the point, it would need to do so by overloading the appropriate function.

The procedural texture must return a list of reflectance function instantiations that represent the procedural texture at that point. This list has a series of reflectance function model instantiation and a weight, allowing for a blend of multiple reflectance functions to be used at the same point.

Figure 6.3 demonstrates the parent class of all procedural textures. Notice the procedural reduction map will have access to all reflectance function model instantiations through this interface. And since the reflectance function models can give exact weights and basis reflectance functions, there is no need for the basis reflectance function identification step using NLS. Furthermore, since the shader normal is given to us, that calculation is no longer needed, either. The Displacement-Free Assumption is obviously alleviated, too (although we do not anti-alias it.)

Although other parameters can be added to those calls, figure 6.3 is for the following

```
class ParentTexture {
public:
    ParentTexture() {}
    virtual ~ParentTexture() {}

    // Perturb the normal. Default returns N.
    virtual Normal PerturbN( const Point &P, const Normal &N );

    // Determine the reflectance function (RF) at point P.
    //  Returns a list of RFModels with corresponding weights.
    virtual void DetermineRF( const Point &P,
        List< RFModel > &RFs, List< float > &weights ) = 0;

    // Determine the displacement mapping. Default returns P.
    virtual Point PerturbP( const Point &P );

    // Render the procedural texture.
    Color Render( const Vector &IncomingLight,
        const Vector &OutgoingLight, const Normal &N,
        const Point &P, float LightIntensity )
    {
        List< RFModel > RFs;
        List< float > weights;

        Point p = PerturbP( P );
        Normal n = PerturbN( p, N );

        DetermineRF( p, RFs, weights );

        Color color = BLACK;
        for ( int i = 0; i < RFs.size(); ++i )
            color += weights[i] * RFs[i]->Intensity(
                        IncomingLight, OutgoingLight, n );
        return color;
    }

private:
};
```

Figure 6.3: The Parent Class of a Procedural Texture.

illustrative purposes.

6.2.3 Removing Other Assumptions

The remaining assumptions are the Light Properties, Distant Light, Non-Deformable Model, and Time Invariance. Since the Light Properties Assumption is vital, there will be no effort to remove that, and since the Distant Light Assumption is common, it shouldn't be a problem, either.

Since the Non-Deformable Model Assumption is not related to the procedural texture and has been previously addressed in section 4.7.5, that topic is not relevant here.

The Time-Invariance Assumption could be easily lifted by adding another virtual function that specifies the units of time the procedural texture uses as well as the length of the loop (looping is required.)

Future advances, such as implementation of BSSRDF rendering, would probably also require more virtual functions to give the procedural reduction map enough information.

6.3 GPU Implementation Details

During rendering using NBRDFs, we want to minimize both the number of texture calls as well as the number of instructions that are executed, thereby having a maximum frames per second. For the most part, there is a small constant number of texture calls in the procedural reduction map algorithm, and the total number of instructions is relatively small.

Creation of the procedural reduction maps and the NBRDFs follow the previous chapters' implementation details. The only difference is in how the procedural reduction map data is stored.

Typically,[1] the procedural reduction map will be stored as three textures and the NBRDFs as a total of two textures. There are three texture calls for evaluating the procedural reduction map and four for the NBRDFs, resulting in seven texture calls for the algorithm.

[1] Sizes will vary depending on the number of basis functions. This will be discussed later.

6.3.1 GPU Procedural Reduction Map

The procedural reduction map has three textures: one for the normal distribution map, one for the specular reflectance functions and one for the diffuse reflectance functions. The normal distribution map texture has four channels, the diffuse three, and the specular has three or four, depending on the number of different specular reflectance functions there are. The procedural reduction map algorithm starts in the fragment shader with the (u, v) coordinates as well as an estimate of the texture footprint.

The normal distribution map stores the average normal and the size of the distribution, to be used as input for the associated NBRDFs. One byte for the size, and three for the normal direction, described by polar coordinates. Polar coordinates are described by two angles, θ and ϕ, each one and a half bytes. Note that the distribution size should correspond to the method for MIP-Map lookups, which is often non-linear.

The diffuse texture stores the color channel weights for the diffuse NBRDF. If multiple diffuse reflectance functions exist in the procedural texture, then multiple diffuse textures are required, but this is unusual.

The specular texture stores the weights for each of the basis specular reflectance functions. This assumes that the specular color does not change, but has a constant color. If two specular reflectance functions vary only in the color, then each is a basis reflectance function with regards to the procedural reduction map. This allows for the result of a NBRDF look-up (an intensity) to be multiplied by a color constant.

6.3.2 GPU NBRDF

The output of the procedural reduction map - the normal, distribution size, diffuse color weights, and specular weights - are fed into the NBRDF. The NBRDF also takes as input the incoming and outgoing light directions.

Even though there can be several NBRDFs for a procedural reduction map, they are all handled in one of two different ways. Each NBRDF represents either a diffuse or specular portion of a reflectance function.

The diffuse NBRDF can be stored in one MIP-Mapped texture, using one byte per texel.

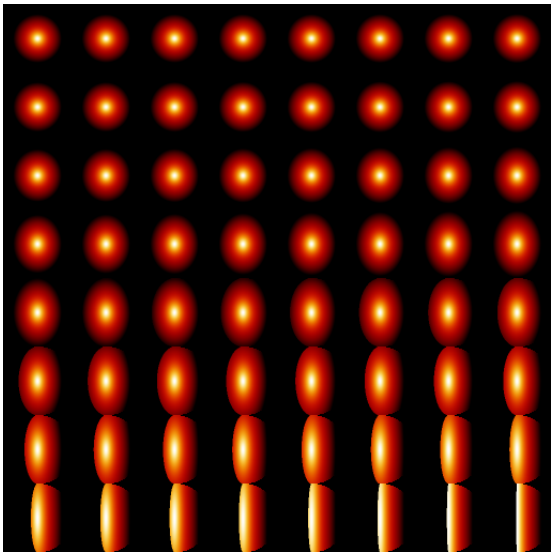


Figure 6.4: Base Storage of the Reflectance Calender. Each color channel stores a different NBRDF. The red channel stores a glossy Phong component and the green and blue channels store a specular Phong component. Notice that the base level of both types of hierarchies (layered and pyramidal) are identical.

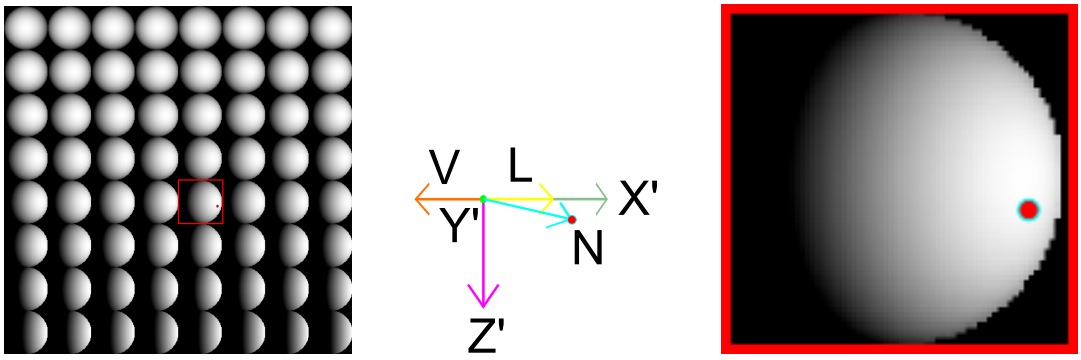
Four specular NBRDFs can be stored in one layered texture using one byte per specular NBRDF. Several specular lookups can therefore be performed simultaneously.

Both types of NBRDFs are the same at the base level. The number of calender slices is set to sixty-four, and the calender slices are arranged in an eight by eight grid (each calender slice is made up of sixty-four by sixty-four texels), as seen in figure 6.4. For the diffuse NBRDF, this is the base of a MIP-Map, and for the specular, it is the base for layered hierarchy eight levels deep.

Inside the vertex shader, the incoming and outgoing light directions are used to compute the two best calender slices for reflectance function integration and the associated weights, as well as the coordinate systems for the calender slice (reference section 5.2.3).

In the fragment shader, using the normal along with the coordinate system (derived from the incoming and outgoing light directions), the coordinates are calculated for use within a reflectance slice. The coordinates are then scaled and shifted to the correct reflectance slice within the luminous calender. See figures 5.15 and 6.5 for more details.

These coordinates are then used to access the NBRDF for the diffuse and specular reflectance functions. Since there are two calender slices (representing the two closest slices



(a) (b) (c)

Figure 6.5: Reflectance Calender Lookup. We want to compute the coordinates to use for texel lookup within the NBRDF reflectance calender. (a) First, we compute which reflectance slice to use within the reflectance calender. We compute this be examining the angle between L and V. In this case, we want to use the reflectance slice outlined by a red square. (b) Within the reflectance slice, we compute the location of the normal (as computed in figure 5.15.) (c) Zooming into the reflectance slice from (a), we apply these coordinates to the reflectance slice to get the final answer (red dot).

to the actual light path), this means four texture lookups. The results are scaled according to the calender slice weights.

The resulting intensities are then scaled by the appropriate colors (for the diffuse NBRDF, the diffuse color weights from the procedural reduction map; and for the specular NBRDF, the specular color times the specular weight from the procedural reduction map.) The summation of these colors is the final result.

6.3.3 GPU Magnification

For reflectance functions whose implementations are simple enough, the original reflectance function model can also be implemented simultaneously. When the distribution size is small enough, the original reflectance function model can be used, providing artifact-free rendering.

Otherwise, specular highlights rendered from NBRDFs can exhibit magnification artifacts. In the case where the original function cannot be used, the lowest level of the NBRDF hierarchy can be smoothed, thereby removing the artifacts, but the severe blurring will impact the visual fidelity.

Furthermore, unlike the procedural reduction map algorithm, the original procedural

texture will often require too many instructions to run in real-time, even on the GPU. However, in spite of this, procedural reduction maps can always run on the GPU in real-time. The downside is the procedural reduction map has finite resolution, and therefore some of the magnification benefits can be lost.

6.3.4 GPU Costs

There are three types of costs for running on the GPU: texture memory, texture calls, and number of instructions. For all procedural textures used in this chapter, all of these were constant, although some effort could have been made to make that constant even smaller for some of these items, and that will be discussed in a moment.

First, the texture memory costs for a procedural reduction map and associated NBRDFs is completely determined by the number of basis reflectance functions that are diffuse and that are specular. The cost of the procedural reduction map (in bytes), p_c, can be seen as:

$$p_c = p_t \times (p_n + p_d + p_s) \tag{6.1}$$

where p_t is the number of texels of the procedural reduction map, p_n is the cost of the normal distribution map, p_d is the cost of the diffuse weights, and p_s is the cost of the specular weights. Typical procedural textures have costs of $p_n = 4$, $p_d = 3$, and $1 \leq p_s \leq 4$.

The cost for the corresponding NBRDFs (in bytes), n_c, is:

$$n_c = b_n \times (n_d + n_s) \tag{6.2}$$

where n_t is the number of texels in a luminous calender, n_d is the cost of the diffuse reflectance functions and n_s is the cost of the specular reflectance functions. Typically, $n_d = 1$, and $n_s = 8 \times s$, where s is the number of different types of specular lobes.

The final cost of the algorithm is usually no more than twelve times the size of corresponding MIP-Map, although it is usually less than that. For certain cases it can be compressed drastically.

Consider the case of an object that is flat, such as a ceiling, floor, or wall. The normal distribution map can be completely left out in this case, dropping p_n to zero.

If all or a portion of the specular lobes are glossy, they can be handled well by the pyramid hierarchical scheme. This would end up dropping the storage cost ($n_s = s \times \frac{4}{3}$) of the specular component drastically.[2]

For a common procedural texture, with one type of diffuse reflectance function (i.e., Phong, Ward, or Ashikhmin, but not a combination) and zero to four different specular lobes (excluding scaling), there are seven texture calls. In certain cases, as delimited previously, this can be reduced. Furthermore, the number of instructions is not too large, and some of which is in the vertex shader providing even more speed.

6.4 Results

The results in this chapter were tested on a dual-processor Intel Xeon, 3.2 GHz machine with 2 GB of memory. The video card was the nVidia Quadro NVS 280 PCI-E with 64 MB of video card memory running a dual monitor system (60 Hz) with each monitor's resolution at 1280x1024, 32 bit.

For purposes of portability, the rendering program was implemented using Sh [70]. The creation program was independent of any system, although it did interface with procedural textures written for the GPU.

Figure 6.6 demonstrates three procedural textures that have interesting characteristics: a bump-mapped procedural texture with high specular highlights applied to a bunny model, a multi-lobed reflectance function applied to a dragon, and a cellular texture applied to the feline model. The figure also demonstrates the result of not using the procedural reduction map for anti-aliasing. For several of the demonstrations, the object is far away and then the resulting image is magnified.

Going from left to right in figure 6.6, the first procedural texture demonstrates a difficult reflectance function, having multi-lobe specular components that normal MIP-Maps cannot handle. The procedural texture is applied to the feline model (10K triangles).

[2] In the case where only one such glossy specular component exists, then it can be combined with the diffuse textures, both in the procedural reduction map algorithm, as well as the NBRDF. This results in three fewer texture calls.

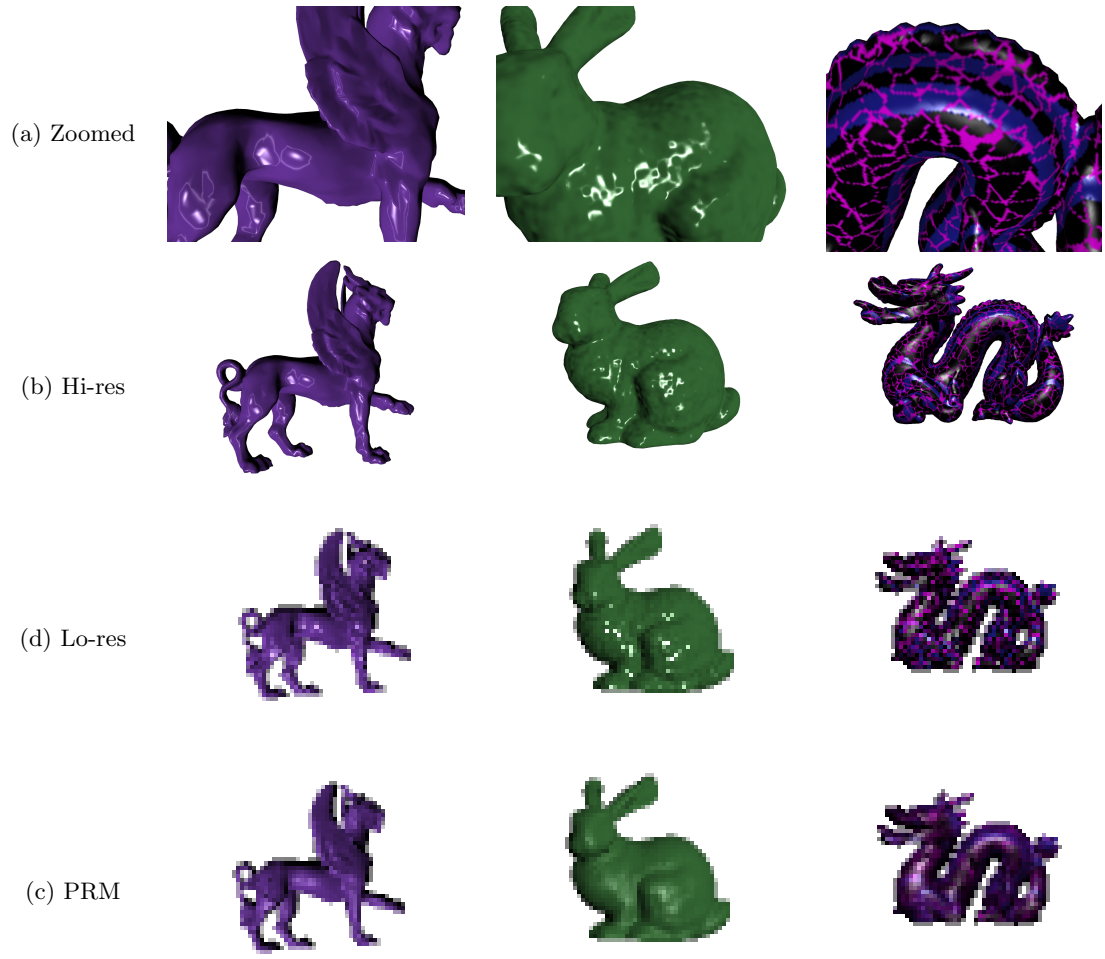


Figure 6.6: GPU Results for Procedural Reduction Maps. (a) A zoomed in view of each procedural texture. (b) A high-resolution picture of the object's texture, seen as a whole. For (c) and (d), the object is far away and then the resulting image is enlarged. (c) The original texture at a distance. Notice that no blurring of the reflectance function has occurred. This results in noticeable aliasing. (d) The procedural reduction map. The highlights are dulled for the bump mapped procedural texture.

Table 6.1: GPU Procedural Reduction Maps Creation Times and Storage Costs. All times are in seconds. The second column states how long rasterization of the object's surface took, and the third column states how long it took to generate the pyramid. The determination of basis reflectance functions and the associated weights make up the remaining portions of the total time (found in the fourth column.) The procedural reduction map storage and the associated NBRDF cost is in the next two columns. The final column states how many texture calls (assuming a MIP-Map texture call in hardware as one texture call.)

Model	Rasterize	Pyramid	Total	Storage (MB)	NBRDF (MB)	Texture Calls
Bunny	17	7	283	2.7	2.3	7
Dragon	17	7	229	3.0	4.6	7
Feline	17	7	61	2.7	2.3	7

The second is the bump-mapped procedural texture with high specular highlights applied to the bunny model (10K triangles). This is difficult to anti-alias on the GPU because of the high frequency as well as the high color disparity between specular and non-specular regions.

The last procedural texture demonstrates cellular textures along with a wide range of reflectance functions applied to the dragon model (20K triangles). Rendering this procedural texture without procedural reduction maps would result in severe degradation of the rendering speed due to the high number of instructions per fragment.

Table 6.1 lists the timings for creation of the procedural reduction map when the authoring guidelines are followed. This does not include any NBRDF creation since they already might be created (see Table 5.1 for those timings.) The table also lists the storage costs for the procedural reduction map used for each procedural texture, along with the storage cost of the associated NBRDFs. Since the equivalent MIP-Map for the same base size is 1 MB, the ratio of using a procedural reduction map compared to a MIP-Map of the same base size is equivalent to the sum of the procedural reduction map and NBRDF storages. The table also lists the number of texture calls required for each procedural reduction map (including the NBRDF calls).

As can be seen from the table, there is a speed increase of several orders of magnitude in procedural reduction map creation. Furthermore, there is an increased accuracy of basis reflectance function identification and instantiation (due to exact normal perturbations.)

The final frame rate for procedural reduction maps on the GPU is roughly one quarter the frame rate of a Phong shader, making procedural reduction maps interactive and nearly real-time on the GPU.

6.5 Conclusions

This chapter presented several goals. Among them were a rasterized procedural reduction map, a broader spectrum of procedural textures covered, and movement onto the GPU.

The rasterization process is straightforward and its evidence is shown through the GPU results. The graphics hardware version of procedural reduction maps runs well and fast through the use of NBRDFs. Although there are seven texture calls for most procedural textures, a few extremely varying (with regard to reflectance functions) procedural textures could require more (i.e., if the number of different specular exponents is greater than 4.)

The storage requirements is not that extravagant, and the frames per second is also fast. Faster GPUs both now and in the future will only increase the speed. In addition, with GPUs that can handle even more instructions, more types of reflectance function models can be processed simultaneously with procedural reduction maps, giving even more capability of magnification benefits.

Procedural reduction maps also handle a broader spectrum of procedural textures through the use of the new texture authoring guidelines. Furthermore, the reflectance function detection issue goes away with these new methods. Despite the increased power of detection, no extra limits were placed on the procedural textures capabilities, making the procedural reduction maps even more desirable.

6.6 Future Work

Due to the constants in the algorithm, the procedural reduction map with NBRDFs might be implementable in hardware, thereby freeing processing power for the magnification aspects. Since much effort is spent on anti-aliasing on graphics hardware, this might prove interesting.

Use of other representations known to work on GPUs, such as homomorphic factorized BRDFs, should be looked into to address magnification and reflectance function integration issues.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

The goal of this research is to perform better minification anti-aliasing of procedural textures through minimal effort on the part of either the texture author or the CPU. Through the use of procedural reduction maps and normal-centric bidirectional reflectance functions, superior results are achieved compared to several current anti-aliasing methods.

A wide range of procedural textures can be used with procedural reduction maps, including all isotropic reflectance functions, bump maps, and in some cases, transparency. Furthermore, the integration of the reflectance functions is superior to most existing methods, providing greater visual fidelity to the original texture's detail.

The render times for the procedural reduction map and the normal-centric bidirectional reflectance function are fast and scale linearly with the number of different types of reflectance functions that exist within the procedural texture. Interactive systems that use the GPU render with at most seven texture calls in almost all cases. Off-line systems, such as a ray tracer, average one sample per pixel, with less than seven procedural texture calls per pixel (often only three to five).

Visual fidelity to the correct anti-aliased result is high due to the better reflectance function integration through the normal-centric bidirectional reflectance function. Although some accuracy was given up for this representation, the loss is small, and the benefit is greatly increased rendering speed.

7.1 Future Work

Future research on procedural reduction maps will benefit from the inherited reflectance function system that allows for detailed information about the reflectance function at any point to be passed to the procedural reduction map creation algorithm. Algorithms that handle displacement mapping, or time-varying textures, or anisotropy can be relatively easy to plug into the procedural reduction map algorithm.

For high-end renderers, such as *RenderMan*, some further work might be necessary to fully use these techniques. Although the creation of a semi-automatic procedural reduction map should be straightforward for any rendering system, the rendering portion will probably require some adjustment. Specifically, RenderMan procedural textures call the renderer to get the texture footprint estimation. The texture then uses this information to do its own minification anti-aliasing. However, procedural reduction maps should sit in-between the renderer and the procedural texture itself. The implementation of the procedural reduction map would have to acquire the texture footprint while maintaining the many parameters that are passed to the original procedural texture. The other option is to have the renderer call the procedural reduction map and then call the procedural texture depending on the output of the procedural reduction map.

Because the input to a MIP-Map is the same as the input to a procedural reduction, the implementation and integration of procedural reduction maps should not be too difficult on any modern rendering system. The NBRDF, uses the input from the procedural reduction map, and and therefore should also be easy to integrate.

Another exciting aspect is the ability to move complex procedural textures onto graphics hardware in an anti-aliased manner. This means that immediate application of this method can be used within the graphics hardware community. For example, video games can have much more complex textures on characters and settings that do not alias when minified. Finally, virtual and augmented reality could also use this for enhanced realism that is relatively inexpensive.

REFERENCES

[1] "The Renderman Interface," *Pixar Animation Studios*, 1989.

[2] Abram, G. D. and Whitted, T., "Building block shaders," in *SIGGRAPH '90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 283–288, ACM Press, 1990.

[3] Amanatides, J., "Ray tracing with cones," in *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 129–135, ACM Press, 1984.

[4] Apodaca, A. A. and Gritz, L., *Advanced RenderMan: Creating CGI for Motion Picture*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[5] Ashikhmin, M. and Shirley, P., "An anisotropic Phong BRDF model," *J. Graph. Tools*, vol. 5, no. 2, pp. 25–32, 2000.

[6] Ashikmin, M., Premože, S., and Shirley, P., "A microfacet-based BRDF generator," in *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 65–74, ACM Press/Addison-Wesley Publishing Co., 2000.

[7] Banks, D. C., "Illumination in diverse codimensions," in *SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 327–334, ACM Press, 1994.

[8] Becker, B. G. and Max, N. L., "Smooth transitions between bump rendering algorithms," in *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 183–190, ACM Press, 1993.

[9] Benson, D. and Davis, J., "Octree textures," in *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 785–790, ACM Press, 2002.

[10] Blinn, J. F., "Models of light reflection for computer synthesized pictures," *SIGGRAPH Comput. Graph.*, vol. 11, no. 2, pp. 192–198, 1977.

[11] Blinn, J. F., "Simulation of wrinkled surfaces," in *SIGGRAPH '78: Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 286–292, ACM Press, 1978.

[12] Blinn, J. F., "Light reflection functions for simulation of clouds and dusty surfaces," *SIGGRAPH Comput. Graph.*, vol. 16, no. 3, pp. 21–29, 1982.

[13] Blinn, J. F. and Newell, M. E., "Texture and reflection in computer generated images," *Commun. ACM*, vol. 19, no. 10, pp. 542–547, 1976.

[14] Blinn, J. F., *Computer Display of Curved Surfaces.* PhD thesis, 1978.

[15] Blythe, D., "The Direct3D 10 system," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, 2006.

[16] Brostow, G. J. and Essa, I., "Image-based motion blur for stop motion animation," in *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 561–566, ACM Press, 2001.

[17] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[18] Cabral, B., Max, N., and Springmeyer, R., "Bidirectional reflection functions from surface bump maps," in *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 273–281, ACM Press, 1987.

[19] Carr, N. A. and Hart, J. C., "Meshed atlases for real-time procedural solid texturing," *ACM Trans. Graph.*, vol. 21, no. 2, pp. 106–131, 2002.

[20] Catmull, E. E., *A Subdivision Algorithm for Computer Display of Curved Surfaces.* PhD thesis, 1974.

[21] Chen, Y., Tong, X., Wang, J., Lin, S., Guo, B., and Shum, H.-Y., "Shell texture functions," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 343–353, 2004.

[22] Cignoni, P., Montani, C., Scopigno, R., and Rocchini, C., "A general method for preserving attribute values on simplified meshes," in *VIS '98: Proceedings of the Conference on Visualization '98*, (Los Alamitos, CA, USA), pp. 59–66, IEEE Computer Society Press, 1998.

[23] Cohen, J., Olano, M., and Manocha, D., "Appearance-preserving simplification," in *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 115–122, ACM Press, 1998.

[24] Cook, R. L. and Torrance, K. E., "A reflectance model for computer graphics," *ACM Trans. Graph.*, vol. 1, no. 1, pp. 7–24, 1982.

[25] Cook, R. L., "Shade trees," *ACM SIGGRAPH*, pp. 223–231, 1984.

[26] Cook, R. L., "Stochastic sampling in computer graphics," *ACM Trans. Graph.*, vol. 5, no. 1, pp. 51–72, 1986.

[27] Cook, R. L. and DeRose, T., "Wavelet noise," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 803–811, 2005.

[28] Cook, R. L., Porter, T., and Carpenter, L., "Distributed ray tracing," in *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 137–145, ACM Press, 1984.

[29] Crow, F. C., *The aliasing problem in computer-synthesized shaded images.* PhD thesis, 1976.

[30] Crow, F. C., "Summed-area tables for texture mapping," in *SIGGRAPH '84: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 207–212, ACM Press, 1984.

[31] Dana, K. J., van Ginneken, B., Nayar, S. K., and Koenderink, J. J., "Reflectance and texture of real-world surfaces," *ACM Trans. Graph.*, vol. 18, no. 1, pp. 1–34, 1999.

[32] Dippé, M. A. Z. and Wold, E. H., "Antialiasing through stochastic sampling," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 69–78, 1985.

[33] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S., *Texturing and Modeling: A Procedural Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[34] Ferrari, L. A., Sankar, P. V., Sklansky, J., and Leeman, S., "Efficient two-dimensional filters using B-spline functions," *Comput. Vision Graph. Image Process.*, vol. 35, no. 2, pp. 152–169, 1986.

[35] Fournier, A., "Normal distribution functions and multiple surfaces," in *Graphics Interface '92 Workshop on Local Illumination*, (Vancouver, BC, Canada), pp. 45–52, 11 May 1992.

[36] Gershbein, R. and Hanrahan, P., "A fast relighting engine for interactive cinematic lighting design," in *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 353–358, ACM Press/Addison-Wesley Publishing Co., 2000.

[37] Goldman, D. B., "Fake fur rendering," in *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 127–134, ACM Press/Addison-Wesley Publishing Co., 1997.

[38] Gortler, S. J., Grzeszczuk, R., Szeliski, R., and Cohen, M. F., "The lumigraph," in *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 43–54, ACM Press, 1996.

[39] Guenter, B., Knoblock, T. B., and Ruf, E., "Specializing shaders," in *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 343–350, ACM Press, 1995.

[40] Hanrahan, P. and Krueger, W., "Reflection from layered surfaces due to subsurface scattering," in *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 165–174, ACM Press, 1993.

[41] Hanrahan, P. and Lawson, J., "A language for shading and lighting calculations," *SIGGRAPH Comput. Graph.*, vol. 24, no. 4, pp. 289–298, 1990.

[42] Hart, J. C., Carr, N., Kameya, M., Tibbitts, S. A., and Coleman, T. J., "Antialiased parameterized solid texturing simplified for consumer-level hardware implementation," in *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, (New York, NY, USA), pp. 45–53, ACM Press, 1999.

[43] Heckbert, P. S., "Filtering by repeated integration," in *SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 315–321, ACM Press, 1986.

[44] Heckbert, P. S., "Survey of texture mapping," *IEEE Comput. Graph. Appl.*, vol. 6, no. 11, pp. 56–67, 1986.

[45] Heidrich, W. and Seidel, H., "Efficient rendering of anisotropic surfaces using computer graphics hardware," 1998.

[46] Heidrich, W. and Seidel, H.-P., "Realistic, hardware-accelerated shading and lighting," in *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 171–178, ACM Press/Addison-Wesley Publishing Co., 1999.

[47] Heidrich, W., Slusallek, P., and Seidel, H.-P., "Sampling procedural shaders using affine arithmetic," *ACM Trans. Graph.*, vol. 17, no. 3, pp. 158–176, 1998.

[48] Hertzmann, A., "Example-based photometric stereo: Shape reconstruction with general, varying BRDFs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 27, no. 8, pp. 1254–1264, 2005. Member-Steven M. Seitz.

[49] Hoppe, H. H., "Progressive meshes," in *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 99–108, ACM Press/Addison-Wesley Publishing Co., 1996.

[50] Igehy, H., "Tracing ray differentials," in *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 179–186, ACM Press/Addison-Wesley Publishing Co., 1999.

[51] Jensen, H. W., Marschner, S. R., Levoy, M., and Hanrahan, P., "A practical model for subsurface light transport," in *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 511–518, ACM Press, 2001.

[52] Kajiya, J. T., "Anisotropic reflection models," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, pp. 15–21, 1985.

[53] Kajiya, J. T., "The rendering equation," in *SIGGRAPH '86: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 143–150, ACM Press, 1986.

[54] Kautz, J., Heidrich, W., and Seidel, H.-P., "Real-time bump map synthesis," in *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, (New York, NY, USA), pp. 109–114, ACM Press, 2001.

[55] Kautz, J. and McCool, M. D., "Interactive rendering with arbitrary BRDFs using separable approximations," in *SIGGRAPH '99: ACM SIGGRAPH 99 Conference Abstracts and Applications*, (New York, NY, USA), p. 253, ACM Press, 1999.

[56] Kautz, J. and McCool, M. D., "Approximation of glossy reflection with prefiltered environment maps," in *Graphics Interface*, pp. 119–126, 2000.

[57] Kautz, J. and Seidel, H.-P., "Towards interactive bump mapping with anisotropic shift-variant BRDFs," in *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, (New York, NY, USA), pp. 51–58, ACM Press, 2000.

[58] Lafortune, E. P. F., Foo, S.-C., Torrance, K. E., and Greenberg, D. P., "Non-linear approximation of reflectance functions," in *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 117–126, ACM Press/Addison-Wesley Publishing Co., 1997.

[59] Latta, L. and Kolb, A., "Homomorphic factorization of BRDF-based lighting computation," in *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 509–516, ACM Press, 2002.

[60] Lawrence, J., Ben-Artzi, A., DeCoro, C., Matusik, W., Pfister, H., Ramamoorthi, R., and Rusinkiewicz, S., "Inverse shade trees for non-parametric material representation and editing," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 735–745, 2006.

[61] Lawrence, J., Rusinkiewicz, S., and Ramamoorthi, R., "Efficient BRDF importance sampling using a factored representation," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 496–505, 2004.

[62] Lawson, C. L. and Hanson, R. J., *Solving Least Squares Problems*. Prentice-Hall Series in Automatic Computation, Englewood Cliffs: Prentice-Hall, 1974, 1974.

[63] Lefebvre, S. and Hoppe, H., "Parallel controllable texture synthesis," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 777–786, 2005.

[64] Liu, X., Yu, Y., and Shum, H.-Y., "Synthesizing bidirectional texture functions for real-world surfaces," in *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 97–106, ACM Press, 2001.

[65] Ma, W.-C., Chao, S.-H., Tseng, Y.-T., Chuang, Y.-Y., Chang, C.-F., Chen, B.-Y., and Ouhyoung, M., "Level-of-detail representation of bidirectional texture functions for real-time rendering," in *SI3D '05: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, (New York, NY, USA), pp. 187–194, ACM Press, 2005.

[66] Malzbender, T., Gelb, D., and Wolters, H., "Polynomial texture maps," in *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 519–528, ACM Press, 2001.

[67] Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J., "Cg: a system for programming graphics hardware in a C-like language," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, 2003.

[68] Matusik, W., Pfister, H., Brand, M., and McMillan, L., "A data-driven reflectance model," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 759–769, 2003.

[69] McAllister, D. K., Lastra, A., and Heidrich, W., "Efficient rendering of spatial bi-directional reflectance distribution functions," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, (Aire-la-Ville, Switzerland, Switzerland), pp. 79–88, Eurographics Association, 2002.

[70] McCool, M. and Du Toit, S., *Metaprogramming GPUs with Sh.* Wellesley, MA, USA: AK Peters, Ltd., 2004.

[71] McCool, M. D., Ang, J., and Ahmad, A., "Homomorphic factorization of BRDFs for high-performance rendering," in *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 171–178, ACM Press, 2001.

[72] Mitchell, D. P., "Generating antialiased images at low sampling densities," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 65–72, 1987.

[73] Mitchell, D. P. and Netravali, A. N., "Reconstruction filters in computer graphics," *SIGGRAPH Comput. Graph.*, vol. 22, no. 4, pp. 221–228, 1988.

[74] Norton, A., Rockwood, A. P., and Skolmoski, P. T., "Clamping: A method of antialiasing textured surfaces by bandwidth limiting in object space," *SIGGRAPH Comput. Graph.*, vol. 16, no. 3, pp. 1–8, 1982.

[75] Olano, M., Hart, J. C., Heidrich, W., and McCool, M., *Real-Time Shading.* Natick, MA: AK Peters, 2002.

[76] Olano, M., Kuehne, B., and Simmons, M., "Automatic shader level of detail," in *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, (Aire-la-Ville, Switzerland, Switzerland), pp. 7–14, Eurographics Association, 2003.

[77] Olano, M. and Lastra, A., "A shading language on graphics hardware: the pixelflow shading system," in *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 159–168, ACM Press, 1998.

[78] Olano, M., Mukherjee, S., and Dorbie, A., "Vertex-based anisotropic texturing," in *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, (New York, NY, USA), pp. 95–98, ACM Press, 2001.

[79] Olano, M. and North, M., "Normal distribution mapping," *UNC Computer Science Technical Report 97-041*, 1997.

[80] Oliveira, M. M., Bishop, G., and McAllister, D., "Relief texture mapping," in *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics*

and Interactive Techniques, (New York, NY, USA), pp. 359–368, ACM Press/Addison-Wesley Publishing Co., 2000.

[81] Peachey, D. R., "Solid texturing of complex surfaces," in *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 279–286, ACM Press, 1985.

[82] Pellacini, F., "User-configurable automatic shader simplification," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 445–452, 2005.

[83] Perlin, K., "State of the art in image synthesis seminar notes," *SIGGRAPH 1985 Course*.

[84] Perlin, K., "An image synthesizer," in *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 287–296, ACM Press, 1985.

[85] Perlin, K., "Improving noise," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 681–682, 2002.

[86] Phong, B. T., "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, no. 6, pp. 311–317, 1975.

[87] Poulin, P. and Fournier, A., "A model for anisotropic reflection," *SIGGRAPH Comput. Graph.*, vol. 24, no. 4, pp. 273–282, 1990.

[88] Ramamoorthi, R. and Hanrahan, P., "An efficient representation for irradiance environment maps," in *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 497–500, ACM Press, 2001.

[89] Ramamoorthi, R. and Hanrahan, P., "Frequency space environment map rendering," in *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 517–526, ACM Press, 2002.

[90] Rioux, M., Soucy, M., and Godin, G., "A texture-mapping approach for the compression of colored 3D triangulations," *The Visual Computer*, vol. 12, no. 10, pp. 503–514, 1996.

[91] Rushmeier, H. E., Taubin, G., and Guéziec, A., "Appying shape from lighting variation to bump map capture," in *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, (London, UK), pp. 35–44, Springer-Verlag, 1997.

[92] Sander, P. V., Snyder, J., Gortler, S. J., and Hoppe, H., "Texture mapping progressive meshes," in *SIGGRAPH '01: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 409–416, ACM Press, 2001.

[93] Schilling, A., Knittel, G., and Strasser, W., "Texram: A smart memory for texturing," *IEEE Comput. Graph. Appl.*, vol. 16, no. 3, pp. 32–41, 1996.

[94] Schlick, C., "An inexpensive BRDF model for physically-based rendering," *Computer Graphics Forum*, vol. 13, no. 3, pp. 233–246, 1994.

[95] Schröder, P. and Sweldens, W., "Spherical wavelets: Efficiently representing functions on the sphere," in *SIGGRAPH '95: Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 161–172, ACM Press, 1995.

[96] Sen, P., "Silhouette maps for improved texture magnification," in *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, (New York, NY, USA), pp. 65–73, ACM Press, 2004.

[97] Shirman, L. A. and Abi-Ezzi, S. S., "The cone of normals technique for fast processing of curved patches.," *Comput. Graph. Forum*, vol. 12, no. 3, pp. 261–272, 1993.

[98] Tan, P., Lin, S., Quan, L., Guo, B., and Shum, H.-Y., "Multiresolution reflectance filtering.," in *Rendering Techniques* (Deussen, O., Keller, A., Bala, K., Dutré, P., Fellner, D. W., and Spencer, S. N., eds.), pp. 111–116, Eurographics Association, 2005.

[99] Taubin, G., "Curve and surface smoothing without shrinkage," *iccv*, vol. 00, p. 852, 1995.

[100] Torrance, K. E. and Sparrow, E. M., "Theory for Off-Specular Reflection from Roughened Surfaces," *Journal of the Optical Society of America (1917-1983)*, vol. 57, pp. 1105–+, Sept. 1967.

[101] Upstill, S., *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Boston, MA: Addison-Wesley Longman Publishing Co., 1989.

[102] Walter, B., Alppay, G., Lafortune, E., Fernandez, S., and Greenberg, D. P., "Fitting virtual lights for non-diffuse walkthroughs," in *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 45–48, ACM Press/Addison-Wesley Publishing Co., 1997.

[103] Ward, G. J., "Measuring and modeling anisotropic reflection," in *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 265–272, ACM Press, 1992.

[104] Westin, S. H., Arvo, J. R., and Torrance, K. E., "Predicting reflectance functions from complex surfaces," in *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 255–264, ACM Press, 1992.

[105] William Dungan, J., Stenger, A., and Sutty, G., "Texture tile considerations for raster graphics," *SIGGRAPH Comput. Graph.*, vol. 12, no. 3, pp. 130–146, 1978.

[106] Williams, L., "Pyramidal parametrics," *SIGGRAPH Comput. Graph.*, vol. 17, no. 3, pp. 1–11, 1983.

[107] Witkin, A. P., "Recovering surface shape and orientation from texture," *Artificial Intelligence*, vol. 17, pp. 17–45, 1981.

[108] Worley, S., "A cellular texture basis function," in *SIGGRAPH '96: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, (New York, NY, USA), pp. 291–294, ACM Press, 1996.

[109] Yao, W.-M., Amsler, C., Asner, D., Barnett, R., Beringer, J., Burchat, P., Carone, C., Caso, C., Dahl, O., D'Ambrosio, G., DeGouvea, A., Doser, M., Eidelman, S., Feng, J., Gherghetta, T., Goodman, M., Grab, C., Groom, D., Gurtu, A., Hagiwara, K., Hayes, K., Hernández-Rey, J., Hikasa, K., Jawahery, H., Kolda, C., Y., K., Mangano, M., Manohar, A., Masoni, A., Miquel, R., Mönig, K., Murayama, H., Nakamura, K., Navas, S., Olive, K., Pape, L., Patrignani, C., Piepke, A., Punzi, G., Raffelt, G., Smith, J., Tanabashi, M., Terning, J., Törnqvist, N., Trippe, T., Vogel, P., Watari, T., Wohl, C., Workman, R., Zyla, P., Armstrong, B., Harper, G., Lugovsky, V., Schaffner, P., Artuso, M., Babu, K., Band, H., Barberio, E., Battaglia, M., Bichsel, H., Biebel, O., Bloch, P., Blucher, E., Cahn, R., Casper, D., Cattai, A., Ceccucci, A., Chakraborty, D., Chivukula, R., Cowan, G., Damour, T., DeGrand, T., Desler, K., Dobbs, M., Drees, M., Edwards, A., Edwards, D., Elvira, V., Erler, J., Ezhela, V., Fetscher, W., Fields, B., Foster, B., Froidevaux, D., Gaisser, T., Garren, L., Gerber, H.-J., Gerbier, G., Gibbons, L., Gilman, F., Giudice, G., Gritsan, A., Grünewald, M., Haber, H., Hagmann, C., Hinchliffe, I., Höcker, A., Igo-Kemenes, P., Jackson, J., Johnson, K., Karlen, D., Kayser, B., Kirkby, D., Klein, S., Kleinknecht, K., Knowles, I., Kowalewski, R., Kreitz, P., Krusche, B., Kuyanov, Y., Lahav, O., Langacker, P., Liddle, A., Ligeti, Z., Liss, T., Littenberg, L., Liu, L., Lugovsky, K., Lugovsky, S., Mannel, T., Manley, D., Marciano, W., Martin, A., Milstead, D., Narain, M., Nason, P., Nir, Y., Peacock, J., Prell, S., Quadt, A., Raby, S., Ratcliff, B., Razuvaev, E., Renk, B., Richardson, P., Roesler, S., Rolandi, G., Ronan, M., Rosenberg, L., Sachrajda, C., Sarkar, S., Schmitt, M., Schneider, O., Scott, D., Sjöstrand, T., Smoot, G., Sokolsky, P., Spanier, S., Spieler, H., Stahl, A., Stanev, T., Streitmatter, R., Sumiyoshi, T., Tkachenko, N., Trilling, G., Valencia, G., van Bibber, K., Vincter, M., Ward, D., Webber, B., Wells, J., Whalley, M., Wolfenstein, L., Womersley, J., Woody, C., Yamamoto, A., Zenin, O., Zhang, J., and Zhu, R.-Y., "Review of Particle Physics," *Journal of Physics G*, vol. 33, pp. 311–313, 2006.

[110] Zhang, E., Mischaikow, K., and Turk, G., "Feature-based surface parameterization and texture mapping," *ACM Trans. Graph.*, vol. 24, no. 1, pp. 1–27, 2005.

[111] Zickler, T. E., Belhumeur, P. N., and Kriegman, D. J., "Helmholtz stereopsis: Exploiting reciprocity for surface reconstruction," *Int. J. Comput. Vision*, vol. 49, no. 2-3, pp. 215–227, 2002.

VITA

Robert Brooks Van Horn III was born on January 24, 1976 at Fairfax Hospital in Northern Virginia. After living in Reston, Virginia for less than a year, his family (Robert Brooks Van Horn, Jr. and Karen Norman Van Horn) moved to Herndon, Virginia, where he would grow up along with his younger sister, Katherine Darby.

Brooks attended Thomas Jefferson High School for Science and Technology, the Governor's School for Science and Technology. During his senior year, he constructed a scanning tunneling microscope, capable of mapping the "surface" of molecules, for his senior science project.

From there, Brooks went on to attend Baylor University, where he majored in computer science. Along with gaining Upsilon Pi Epsilon computer science honors, he went on to graduate "with Honors with Distinction" in December of 1998. Outside of school, he also started and co-led a student organization with over three hundred members.

For a year and a half, Brooks continued his computer science studies at Baylor in their masters program. Although he did not finish there, Brooks gained valuable experience in programming under the tutelage of David Sturgill.

In the fall of 2000, Brooks entered the Ph.D. program in computer science at Georgia Institute of Technology. Greg Turk became his advisor that year and guided Brooks through his years of graduate study. He finished his Masters studies along the way and worked two summer internships at Pixar Animation Studios.

INDEX